Instituto de Computação
Universidade Estadual de Campinas

O Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional

Pedro Henrique Del Bianco Hokama

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Pedro Henrique Del Bianco Hokama e aprovada pela Banca Examinadora.

Campinas, 14 de Outubro de 2011.

Flávio Keidi Miyazawa (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA POR ANA REGINA MACHADO – CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA – UNICAMP

H689p

Hokama, Pedro Henrique Del Bianco, 1986-
O problema do caixeiro viajante com restrições de empacotamento tridimensional / Pedro Henrique Del Bianco Hokama. – Campinas, SP : [s.n.], 2011.

Orientador: Flávio Keidi Miyazawa.
Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

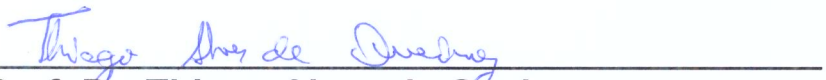
1. Problema do caixeiro viajante. 2. Programação por restrições. 3. Otimização combinatória. 4. Problema do corte de estoque. 5. Programação inteira. I. Miyazawa, Flávio Keidi, 1970-. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Informações para Biblioteca Digital

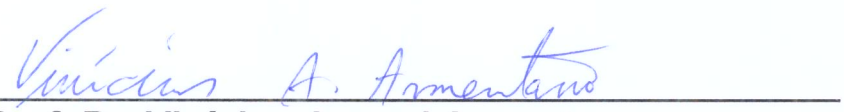
Título em inglês: The traveling salesman problem with three-dimensional loading constraints
Palavras-chave em inglês:
Traveling-salesman problem
Constraint programming (Computer science)
Combinatorial optimization
Cutting stock problem
Integer programming
Área de concentração: Ciência da Computação
Titulação: Mestre em Ciência da Computação
Banca examinadora:
Flávio Keidi Miyazawa [Orientador]
Vinícius Amaral Armentano
Thiago Alves de Queiroz
Data da defesa: 14-10-2011
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 14 de outubro de 2011, pela Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Thiago Alves de Queiroz
Departamento de Matemática / UFG



Prof. Dr. Vinícius Amaral Armentano
FEEC / UNICAMP

Prof. Dr. Flávio Keidi Miyazawa
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

O Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional

Pedro Henrique Del Bianco Hokama[1]

Outubro de 2011

Banca Examinadora:

- Flávio Keidi Miyazawa (Orientador)
- Vinícius Amaral Armentano
  Faculdade de Engenharia Elétrica e de Computação - UNICAMP
- Thiago Alves de Queiroz
  Departamento de Matemática - UFG/CAC
- Yoshiko Wakabayashi (Suplente)
  Instituto de Matemática e Estatística - USP
- Orlando Lee (Suplente)
  Instituto de Computação - UNICAMP

[1]Projeto Financiado pela Fapesp (processo 2009/13270-0) 2010–2011

Resumo

Nesta dissertação de mestrado apresentamos um método exato para o Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional, que combina o Problema do Caixeiro Viajantee o Problema de Empacotamento Tridimensional com Restrição de Ordem. Neste problema, um veículo deve partir carregado de um depósito e entregar caixas em pontos pré-definidos para seus clientes. Cada cliente tem um conjunto de caixas que deve receber e o objetivo é minimizar o custo de deslocamento do veículo. As caixas devem ser retiradas a partir da porta do contêiner do veículo e a remoção das caixas de um cliente não podem ser obstruídas pelas caixas a serem descarregadas posteriormente.

Propomos uma abordagem exata baseada em *branch-and-cut* para buscar uma rota de custo mínimo. Apresentamos algumas adaptações de algoritmos da literatura e uma formulação em Programação por Restrições para encontrar um empacotamento que obedece restrições de ordem. Realizamos testes computacionais em instâncias geradas aleatoriamente e comparamos resultados com os algoritmos adaptados da literatura. Os resultados foram bastante satisfatórios resolvendo instâncias de tamanho médio em tempo computacional aceitável na prática.

Abstract

We present an exact method for the *Traveling Salesman Problem with Three-dimensional Loading Constraints*. This problem combines the *Traveling Salesman Problem*, and the *Three-Dimensional Packing Problem With Loading Constraints*. In this problem, a vehicle must be loaded at the depot and deliver boxes to the customers. Every customer has a set of boxes that should receive and our goal is to minimize the travel cost of the vehicle. Unloading is done through a single side of the container and items from an unloading customer must not be blocked by items to be delivered later.

We propose exact and heuristic branch-and-cut algorithm to find a minimum cost route. Adaptations of algorithms from the literature and a Constraint Programming formulation is presented to find a packing that consider unloading contraints. We performed computational tests on instances randomly generated, and compared results with the algorithms adapted from literature. The results were quite satisfactory resolving several instances in reasonable computational time.

Agradecimentos

Agradeço primeiramente à minha família por todo o suporte ao longo dos anos.

À Priscila por toda a parceria, que torna tudo mais fácil.

Ao pessoal do LOCo que fazem todos os dias divertidos.

Aos meus amigos de longa data, Chineis, Xein e Fábio. Mesmo longe, continuam perto.

A todo povo brasileiro que contribui arduamente com seus impostos.

Agradeço a todo o pessoal da Secretaria do IC, que sempre dá todo o apoio que precisamos.

Agradeço ao Flávio, por toda a orientação, paciência e dedicação.

Agradeço ao CNPq, à CAPES e todas as agências de fomento, em especial à FAPESP que acreditou neste trabalho.

Sumário

Resumo **v**

Abstract **vi**

Agradecimentos **vii**

1 Introdução **1**
1.1 Objetivos do trabalho 1
1.2 Organização do texto 2

2 Descrição do Problema **3**
2.1 Sub-problemas 3
2.1.1 Caixeiro Viajante 3
2.1.2 Empacotamento Tridimensional 6
2.1.3 Empacotamento Tridimensional com Restrição de Ordem 9
2.2 Caixeiro Viajante com Empacotamento Tridimensional 10
2.3 Revisão Bibliográfica 13

3 Técnicas **15**
3.1 Branch-and-bound 15
3.2 Planos de Corte 18
3.2.1 Branch-and-cut 21
3.3 Programação por Restrições 21
3.3.1 Satisfação de Restrições 22

4 Algoritmos para o Problema de Empacotamento Tridimensional **24**
4.1 Algoritmo Exato para o Empacotamento por Robô 24
4.2 Algoritmo Exato baseado em programação por restrições 33
4.3 Uma formulação através de Programação por Restrições 37
4.3.1 Restrição de ordem 38

4.3.2 Redução de Domínio 39
4.4 Heurísticas para o 3DPLC 40
4.4.1 Heurística de George e Robinson 41
4.4.2 Heurística de Empacotamento em Níveis 43

5 Um Algoritmo Exato para o 3DTSPL **46**
5.1 Formulação para o 3DTSPL 46
5.2 Restrições de Conectividade 48
5.2.1 Corte Mínimo 48
5.2.2 Árvore de Gomory-Hu 49
5.3 Restrições de Empacotamento 52
5.4 Rotina de separação para o 3DTSPL 53

6 Resultados Computacionais **55**
6.1 Resultados para o Problema de Empacotamento 56
6.2 Resultados para o 3DTSPL 58

7 Conclusões **64**

Bibliografia **66**

Lista de Tabelas

2.1 (a) Empacotamento por Robô e (b) Empacotamento Geral. 8

6.1 Experimentos Computacionais: Caso irrestrito. 56
6.2 Experimentos Computacionais: Comparação entre o CP, MPV e o MPVBK. . . 57
6.3 Experimentos Computacionais: comparação entre CP e MPVBK para o 3DTSPL. 59
6.4 Experimentos Computacionais: comparação do uso de heurísticas para o 3DTSPL. 60
6.5 Experimentos Computacionais: instâncias da literatura adaptadas ao 3DTSPL. . 61

Lista de Figuras

2.1 A figura 2.1a representa o grafo do jogo do dodecaedro que é hamiltoniano. O grafo bipartido com número ímpar de vértices na figura 2.1b não é hamiltoniano 4
2.2 Circuito Hamiltoniano de Custo mínimo. 4
2.3 Subciclos formados pela ausência de restrições de conectividade. 6
2.4 Exemplo de contêiner e caixa. 7
2.5 A figura 2.5a representa o empacotamento apresentado na tabela 2.1a, e a figura 2.5b o empacotamento apresentado na tabela 2.1b. 9
2.6 Região hachurada corresponde a $\mathcal{R}^z(i)$. 10
2.7 Exemplo de rota com os itens de cada cliente e um empacotamento que respeita a ordem da rota. 12

3.1 Árvore de subproblemas. 16
3.2 Poda por limitante. 17
3.3 Poda por otimalidade. 17
3.4 Poda por Inviabilidade. 18
3.5 Solução ótima para o problema 3.8. 19
3.6 Solução ótima com a adição da desigualdade. 20
3.7 Solução ótima inteira. 20

4.1 (a) Um exemplo de empacotamento bidimensional e (b) seu respectivo envelope. 25
4.2 Exemplo de pontos de canto (2D) [24]. 26
4.3 Exemplo de pontos de canto (3D). 29
4.4 Novos espaços gerados. 42
4.5 Prateleira, Camada e Contêiner. 44

5.1 Exemplo de corte entre E e A. 49
5.2 Um exemplo de construção de Árvore de Gomory-Hu. 52

6.1 Experimentos Computacionais: Profundidade e Custo da Rota - He-76-29-7 . . 62
6.2 Experimentos Computacionais: Profundidade e Custo da Rota - He-76-29-29 . 62
6.3 Experimentos Computacionais: Profundidade e Custo da Rota - Ho-78-37-11 . 63

6.4 Experimentos Computacionais: Profundidade e Custo da Rota - Ho-78-37-37 . 63

Capítulo 1

Introdução

O transporte é uma das principais e mais importantes etapas da logística, sendo responsável por uma grande fatia dos custos embutidos nas mercadorias. Em um mercado globalizado e extremamente competitivo, diminuir os custos envolvidos nessa etapa da distribuição é mandatório para qualquer empresa. Um problema bastante comum em diversas empresas, consiste em entregar um conjunto de caixas em locais diferentes, e em cada um desses locais a entrega deve ser feita de forma eficiente, não sendo viável retirar varias caixas que serão descarregadas posteriormente, para alcançar as caixas da entrega atual.

Nesse trabalho apresentamos o Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional. Esse problema é *NP*-difícil pois é uma generalização do Problema do Caixeiro Viajante, e do Problema de Empacotamento Tridimensional, ambos *NP*-difíceis, e descritos com mais detalhes do decorrer dessa dissertação.

Propomos uma abordagem exata baseada em *branch-and-cut* para buscar uma rota de custo mínimo. Adaptações de algoritmos da literatura e uma formulação em Programação por Restrições são apresentadas para encontrar um empacotamento que obedece restrições de ordem. Além disso, duas heurísticas para este problema também são apresentadas. Também foi considerada uma variante do Problema de Empacotamento Tridimensional, onde o empacotamento deve ser possível de ser realizado por um robô, denominada Problema de Empacotamento por Robô.

1.1 Objetivos do trabalho

O objetivo desse trabalho é apresentar uma abordagem exata para o Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional. Por ser um problema *NP*-difícil e na prática bastante complicado de se resolver, a maioria dos trabalhos da literatura para problemas semelhantes envolve heurísticas, ou métodos aproximados de resolução. Nossa meta é estudar técnicas que levem a soluções exatas e que sejam capazes de resolver instâncias de tamanho

razoável em tempo computacional aceitável na prática.

1.2 Organização do texto

Esta dissertação de mestrado está organizada da forma como segue. No capítulo 2 formalizamos o problema que propomos tratar, alguns sub-problemas e variantes de importância. O capítulo 3, apresenta de maneira geral as técnicas importantes para a descrição dos algoritmos. O capítulo 4 descreve alguns algoritmos para o Problema de Empacotamento Tridimensional. Esses algoritmos serão utilizados como sub-rotina para a abordagem exata, do Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional, proposta no capítulo 5. Os resultados obtidos serão apresentados no capítulo 6. Por fim, algumas conclusões são feitas no capítulo 7.

Capítulo 2

Descrição do Problema

Neste capítulo descrevemos formalmente o Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional (*Traveling Salesman Problem with Three-dimensional Loading Constraints* - 3DTSPL) que propomos a tratar. Também descrevemos alguns sub-problemas, como o Problema do Caixeiro Viajante (*Traveling Salesman Problem* - TSP) e o Problema de Empacotamento Tridimensional com Restrição de Ordem (*Three-Dimensional Packing Problem With Loading Constraints* - 3DPLC), que são importantes para a formalização do 3DTSPL. Por fim, apresentaremos alguns dos principais trabalhos da literatura que estão relacionados com este estudo.

2.1 Sub-problemas

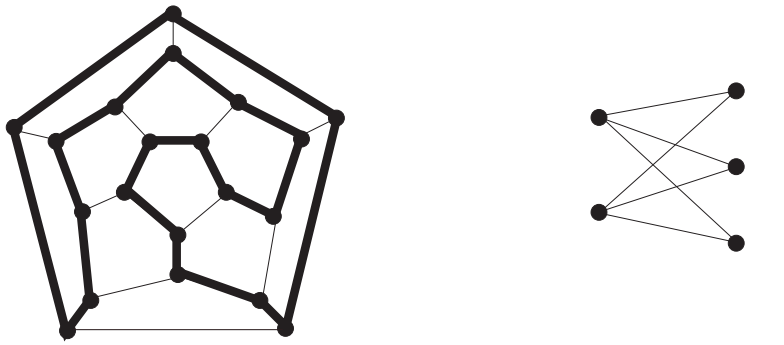
Iniciamos o capítulo introduzindo os três principais sub-problemas envolvidos nesse trabalho. O Problema do Caixeiro Viajante, o Problema de Empacotamento Tridimensional e o Problema de Empacotamento Tridimensional com Restrição de Ordem, são apresentados nas seções 2.1.1, 2.1.2 e 2.1.3, respectivamente.

2.1.1 Caixeiro Viajante

O Problema do Caixeiro Viajante é um dos problemas mais investigados na pesquisa operacional. Esse fenômeno provavelmente se deve a grande utilidade prática, a simplicidade com que pode ser descrito e ao mesmo tempo que é tão desafiador de se resolver. Neste problema um vendedor tem um conjunto de cidades que deve visitar exatamente uma vez e então retornar ao ponto de partida, devendo no entanto fazer esse percurso com o menor custo possível. Para formalizar o problema, primeiramente precisaremos definir o conceito de *circuito hamiltoniano*, essencial para a descrição do Problema do Caixeiro Viajante.

Definição 2.1.1. Dado um grafo não orientado $G = (V, E)$, um circuito hamiltoniano é uma permutação $(v_1, \ldots, v_n)$ de V, tal que $\{v_i, v_{i+1}\} \in E$ para $i = 1, \ldots, n-1$, e $\{v_n, v_1\} \in E$. Se G contém um circuito hamiltoniano então dizemos que G é um grafo hamiltoniano.

Decidir se um dado grafo G é hamiltoniano é um problema NP-completo. A figura 2.1a mostra o grafo utilizado no jogo do dodecaedro, enviado por Sir William Rowan Hamilton ao seu amigo Graves em 1856, no qual um participante coloca cinco alfinetes em cinco vértices consecutivos, e o outro participante deve completar o caminho para formar um circuito contendo todos os vértices. As arestas grossas indicam um circuito hamiltoniano.



(a) Dodecaedro (b) Não Hamiltoniano

Figura 2.1: A figura 2.1a representa o grafo do jogo do dodecaedro que é hamiltoniano. O grafo bipartido com número ímpar de vértices na figura 2.1b não é hamiltoniano

Um segundo conceito bastante importante, é o conceito de *circuito hamiltoniano de custo mínimo*. Para defini-lo considere que cada aresta e tem um custo associado c_e, e queremos encontrar um circuito hamiltoniano $\mathcal{C}$ de custo mínimo. A definição é dada a seguir.

Definição 2.1.2. Dado um grafo $G = (V, E)$, e uma função de custos $c : E \to \mathbb{Q}^+$, onde $\mathbb{Q}^+$ é o conjunto dos números racionais positivos, um circuito hamiltoniano de custo mínimo, é um circuito hamiltoniano $\mathcal{C}$ tal que $\sum_{e \in \mathcal{C}} c_e$ seja mínimo

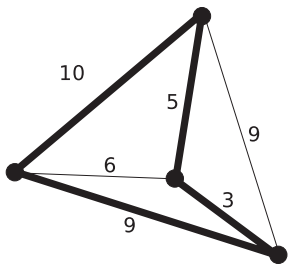


Figura 2.2: Circuito Hamiltoniano de Custo mínimo.

A figura 2.2 mostra um grafo com custos nas arestas e mostra destacado um circuito hamiltoniano de custo mínimo, os valores representam os custos. Podemos então definir o Problema do Caixeiro Viajante utilizando os conceitos apresentados anteriormente.

Problema do Caixeiro Viajante (TSP)**:** *São dados:*
(i) *um grafo não orientado,* $G = (V, E)$,
(ii) *uma função de custos nas arestas* $c : E \to \mathbb{Q}^+$.

O objetivo consiste em encontrar um circuito hamiltoniano de custo mínimo.

Usando a notação acima, denotaremos por $\mathcal{I}_{\text{TSP}}$ o conjunto de tuplas (V, E, c) que são instâncias do Problema do Caixeiro Viajante, onde V é um conjunto de vértice, E um conjunto de arestas e c uma função de custos nas arestas. Seja S um subconjunto próprio de V, denotamos por $\delta(S)$ o conjunto de todas as arestas que incidem somente em um vértice de S, ou seja $\delta(S) = \{u, v : u \in S \text{ e } v \notin S\}$. Simplificaremos a notação $\delta(\{v\})$ para $\delta(v)$, quando o conjunto S for formado por apenas um vértice. Nesse caso, $\delta(v)$ é o conjunto de todas as arestas que incidem em v.

A seguir apresentamos a formulação em Programação Linear Inteira considerada para o TSP. Seja x_e a variável binária que recebe 1 se a aresta e pertence a solução e 0 caso contrário.

$$\text{Minimize} \sum_{e \in E} c_e x_e$$

sujeito a

$$\sum_{e \in \delta(v)} x_e = 2 \qquad \forall v \in V\,, \tag{2.1}$$

$$\sum_{e \in \delta(S)} x_e \geq 2 \qquad \forall S \subset V, \quad S \neq \emptyset\,, \tag{2.2}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E\,. \tag{2.3}$$

A classe de restrições (2.1) indica que, para cada vértice $v \in V$, exatamente duas arestas que incidem em v devem pertencer a solução. A classe de restrições (2.3) garante que as variáveis tenham apenas valor 0 ou 1. Somente estas duas restrições não são suficientes para garantir uma solução correta, pois não garantem a conectividade da solução, podendo permitir a formação de subciclos. A figura 2.3 apresenta uma atribuição onde as restrições (2.1) e (2.3) são atendidas, porém não é uma solução para o Problema do Caixeiro Viajante. As arestas pontilhadas são aquelas em que $x_e = 0$, as arestas contínuas representam que $x_e = 1$.

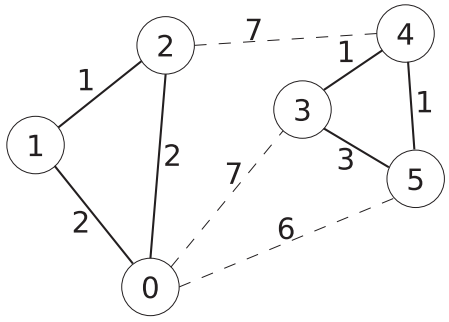


Figura 2.3: Subciclos formados pela ausência de restrições de conectividade.

A classe de restrições (2.2) é necessária para evitar subciclos, garantindo a conectividade da solução. A quantidade de restrições dessa classe é exponencial no número de vértices. Veremos com mais detalhes no capítulo 5.

2.1.2 Empacotamento Tridimensional

Informalmente, o Problema de Empacotamento Tridimensional consiste em encontrar uma forma de alocar um conjunto de caixas em um recipiente, chamado de *contêiner*. Este problema possui diversas aplicações, principalmente no setor de transporte e logística. Ele também pode ser visto como o problema de se obter itens pequenos a partir de algum material em forma de paralelepípedo, através de cortes. Esse caso é comum na produção de peças de madeira, ou espuma de alta densidade. A seguir formalizamos o problema, e depois apresentamos algumas variantes que são comumente encontradas na literatura.

Dado um contêiner B e uma lista de caixas L, em que B tem dimensões (W, H, D), respectivamente a largura, altura e a profundidade, com uma abertura na frente de dimensões (W, H). Cada caixa $i \in L$ tem dimensões (w_i, h_i, d_i) que são, respectivamente, a largura, altura e profundidade da caixa (veja figura 2.4). Um *empacotamento ortogonal* de L em B é uma alocação de cada caixa $i \in L$ em B, tal que as caixas não podem se sobrepor nem ultrapassar os limites do contêiner. Além disso considerando o espaço $\mathbb{R}^3$, a largura de cada caixa e do contêiner devem estar paralelas ao eixo X, as alturas paralelas ao eixo Y e as profundidades paralelas ao eixo Z, e rotações prévias dos itens não são permitidas. Neste trabalho, sempre nos referimos a um empacotamento ortogonal ao usar o termo *empacotamento*.

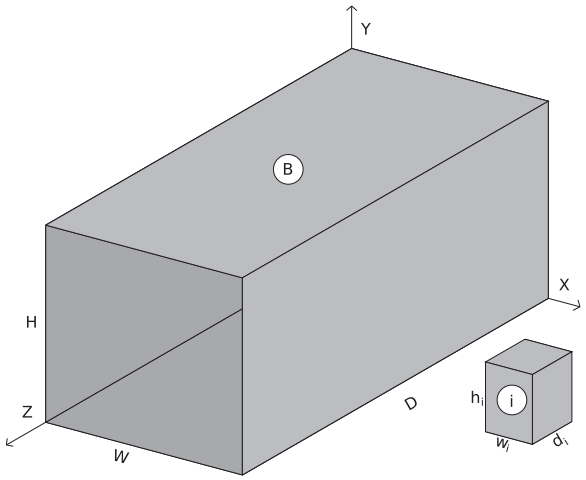


Figura 2.4: Exemplo de contêiner e caixa.

Definição 2.1.3. Um empacotamento de L no contêiner B é uma função $\mathcal{P} : L \to [0, W) \times [0, H) \times [0, D)$, que atribui uma posição no contêiner para cada caixa, correspondente ao canto inferior, esquerdo e traseiro da caixa. Ou seja, definimos para cada caixa $i \in L$, $\mathcal{P}(i) = (x_i, y_i, z_i)$, sua posição dentro do contêiner. O ponto $(0, 0, 0)$ corresponde ao canto inferior, esquerdo e traseiro do contêiner. Dessa forma, podemos definir um empacotamento tal que:

1. Nenhum item passa dos limites do contêiner. Isto é,

$$x_i + w_i \leq W, \;\; y_i + h_i \leq H \;\; \text{e} \;\; z_i + d_i \leq D.$$

2. Dois itens não podem se sobrepor. Assim, para $i \in L$, definimos $\mathcal{R}(i)$ como a região ocupada por i

$$\mathcal{R}(i) = [x_i, x_i + w_i) \times [y_i, y_i + h_i) \times [z_i, z_i + d_i),$$

então

$$\mathcal{R}(i) \cap \mathcal{R}(j) = \emptyset \;\; \forall i, j \in L, \; i \neq j.$$

Dada a definição de um empacotamento, podemos então definir o Problema de Empacotamento Tridimensional. Nesse problema recebemos uma lista de caixas, e desejamos encontrar um empacotamento dessa lista em um contêiner.

Problema de Empacotamento Tridimensional (3DP)**:** *São dados:* (i) *uma lista de caixas L, cada caixa $i \in L$ com dimensões (w_i, h_i, d_i) e* (ii) *um contêiner B de dimensão (W, H, D).* *O objetivo consiste em encontrar um empacotamento das caixas de L em B.*

Denotaremos o conjunto de instâncias $\mathcal{I}_{\text{3DP}}$, do problema 3DP, como o conjunto das tuplas (B, L), onde B é um contêiner de dimensões (W, H, D) e L é uma lista de caixas, sendo cada caixa $i \in L$ com dimensões (w_i, h_i, d_i).

Empacotamento por Robôs

Um empacotamento por robô é um empacotamento que pode ser obtido colocando sucessivamente os itens em um contêiner, iniciando por aquele que não possui qualquer item a sua direita, atrás ou abaixo. Continuando de forma que cada item é colocado em frente, a direita ou acima de cada um dos itens já alocados. Denominamos o problema de se encontrar um empacotamento tridimensional com essas características, por Problema de Empacotamento por Robô (*Robot Packing Problem* - RPP).

Na prática esta restrição ocorre quando o empacotamento dos itens é feito por uma mão mecânica paralela a base do contêiner. Para evitar colisões, nenhum item já alocado pode estar em frente, a direita ou acima do novo item a ser empacotado.

O exemplo a seguir, apresentado por den Boef, Korst, Martello, Pisinger e Vigo [8], mostra na figura 2.5a e na tabela 2.1a um empacotamento possível de ser realizado por um robô. Claramente um empacotamento por robô é uma solução para o empacotamento geral, porém nem todo empacotamento geral pode ser realizado por robô. A figura 2.5b e a tabela 2.1b apresentam uma instância que não possui solução no Problema de Empacotamento por Robô, mas sim no caso geral. As tabelas 2.1a e 2.1b apresentam as dimensões de cada caixa i, (w_i, h_i, d_i) e a posição em que ela é empacotada (x_i, y_i, z_i). O tamanho do contêiner é $(9, 9, 9)$.

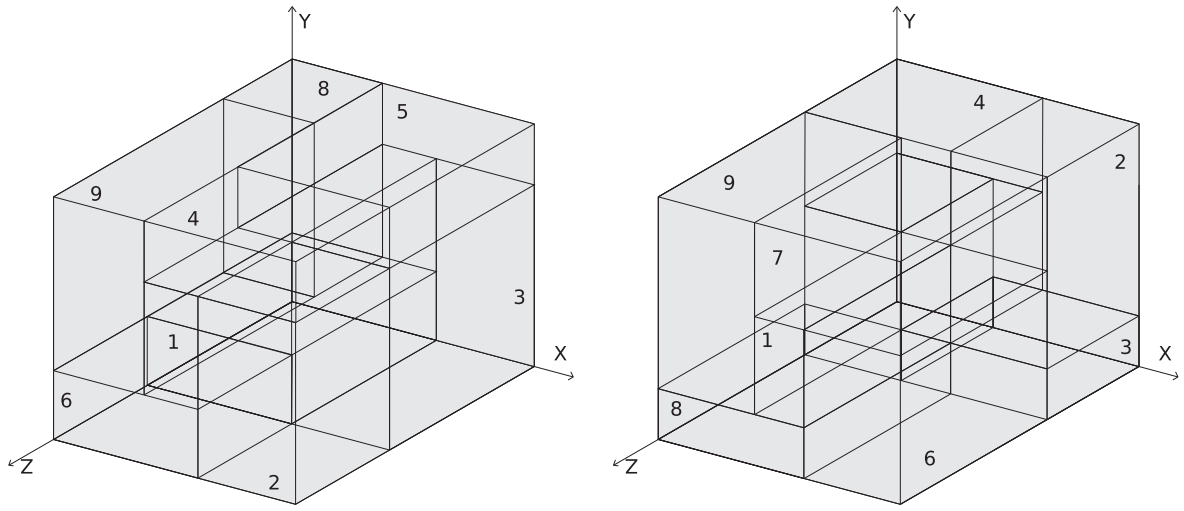
i	w_i	h_i	d_i	x_i	y_i	z_i
1	2	4	9	4	3	0
2	3	7	3	6	0	6
3	3	7	6	6	0	0
4	5	2	3	4	7	6
5	5	2	6	4	7	0
6	6	3	3	0	0	6
7	6	3	6	0	0	0
8	4	6	3	0	3	0
9	4	6	6	0	3	3

(a) Robô

i	w_i	h_i	d_i	x_i	y_i	z_i
1	2	4	9	4	2	0
2	3	7	4	6	2	0
3	5	2	4	4	0	0
4	6	3	4	0	6	0
5	4	6	4	0	0	0
6	3	6	5	6	0	4
7	5	3	5	4	6	4
8	6	2	5	0	0	4
9	4	7	5	0	2	4

(b) Geral

Tabela 2.1: (a) Empacotamento por Robô e (b) Empacotamento Geral.



(a) Robô (b) Geral

Figura 2.5: A figura 2.5a representa o empacotamento apresentado na tabela 2.1a, e a figura 2.5b o empacotamento apresentado na tabela 2.1b.

Abordamos esse problema com mais detalhes na seção 4.1, em que apresentamos um algoritmo *branch-and-bound* exato para o Problema de Empacotamento por Robô.

2.1.3 Empacotamento Tridimensional com Restrição de Ordem

Em muitas aplicações de transporte de itens tridimensionais em contêineres é necessário que o descarregamento dos itens seja feito em determinada ordem, sem que os itens a serem descarregados posteriormente bloqueiem a saída do item atual. Descrevemos a seguir o Problema de Empacotamento Tridimensional com Restrição de Ordem, onde estamos interessados em empacotamentos com essa característica.

Seja $\mathcal{L} = (L_1, L_2, \ldots, L_k)$ uma sequência de listas de caixas e B um contêiner de dimensões (W, H, D), a seguir definimos um empacotamento que respeita a ordem de $\mathcal{L}$. Para isso, definimos $\mathcal{R}^z(i) = [x_i, x_i + w_i) \times [y_i, y_i + h_i) \times [z_i, D)$, a região que a caixa ocupa e toda região entre a caixa e a abertura do contêiner. Dizemos, então, que um empacotamento $\mathcal{P}$ de $L = L_1 \cup \ldots \cup L_k$ respeita a sequência de $\mathcal{L}$ na direção z se $\mathcal{R}^z(i) \cap \mathcal{R}(j) = \emptyset$ para todo par i, j tal que $i \in L_u, j \in L_v$ e $u < v$. Isto é, se i é um item que pertence à L_u, não há nenhuma caixa j na frente de i que pertence a uma lista L_v. A figura 2.6 representa hachurada a região $\mathcal{R}^z(i)$ de uma caixa i, essa região não pode conter qualquer item $j \in L_v$, tal que $u < v$.

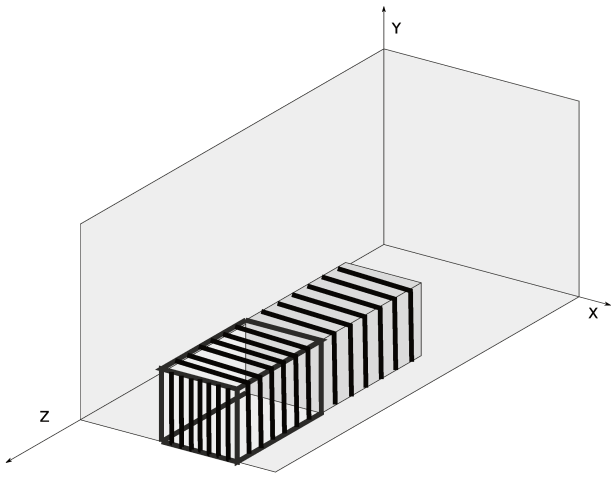


Figura 2.6: Região hachurada corresponde a $\mathcal{R}^z(i)$.

No Problema de Empacotamento Tridimensional com Restrição de Ordem recebemos uma sequência de listas de caixas $\mathcal{L} = (L_1, L_2, \ldots, L_k)$, que devem ser empacotadas em um contêiner B, tal que esse empacotamento respeite a ordem de $\mathcal{L}$.

Problema de Empacotamento Tridimensional com Restrição de Ordem (3DPLC)**:**
São dados:
(i) *uma sequência de listas de caixas* $\mathcal{L} = (L_1, L_2, \ldots, L_k)$,
(ii) *um contêiner* B *de dimensão* (W, H, D).

O objetivo consiste em encontrar um empacotamento $\mathcal{P}$ *das listas de caixas em* $\mathcal{L}$ *em* B*, tal que* $\mathcal{P}$ *respeite a ordem de* $\mathcal{L}$.

Denotamos o conjunto de instâncias $\mathcal{I}_{\text{3DPLC}}$ do problema 3DPLC como o conjunto das tuplas $(B, \mathcal{L})$, onde B é um contêiner de dimensões (W, H, D), e $\mathcal{L}$ uma sequência de listas de caixas.

2.2 Caixeiro Viajante com Empacotamento Tridimensional

O Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional surge em problemas de logística. Nesse problema há um conjunto de clientes e um depósito. Cada cliente deseja receber alguns itens que estão estocados no depósito. Devemos encontrar uma ordem para atender aos clientes e uma alocação dos itens de cada cliente dentro de um contêiner de forma que, em cada cliente visitado seja possível fazer o descarregamento de seus itens sem a remoção dos itens a serem descarregados posteriormente.

Usando as definições dadas na seção 2.1, podemos formalizar o Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional como segue.

Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional (3DTSPL): *São dados:*
(i) $(V, E, c) \in \mathcal{I}_{\text{TSP}}$, $V = \{0, 1, \ldots, k\}$;
(ii) $(B, \mathbb{L}) \in \mathcal{I}_{\text{3DPLC}}$, *tal que* $\mathbb{L} = \{L_1, \ldots, L_k\}$ *e* $B = (W, H, D)$.

O objetivo consiste em encontrar um circuito hamiltoniano $\mathcal{C} = (0, v_1, \ldots, v_k)$ *de custo mínimo, tal que exista um empacotamento das caixas de* $\mathbb{L}$ *em* B*, que respeite a ordem de* $\mathcal{L} = (L_{v_1}, L_{v_2}, \ldots, L_{v_k})$.

Note que na definição acima é necessário escolher uma direção em que os vértices serão visitados. O empacotamento deve ser realizado conforme descrito na seção 2.1.3. Cada diferente circuito hamiltoniano será chamado de rota e representa a ordem com que serão visitados os clientes.

A figura 2.7 exemplifica um empacotamento que respeita ordem para a rota $(0, 1, 2, 3, 4)$. Os itens que cada cliente receberá estão indicados pelo número em cada item. A retirada dos itens é realizada no sentido da seta pontilhada.

Denotamos o conjunto de instâncias $\mathcal{I}_{\text{3DTSPL}}$ do problema 3DTSPL como o conjunto das tuplas $(V, E, c, B, \mathbb{L})$, onde $G = (V, E, c)$ é o grafo de entrada, $B = (W, H, D)$ as dimensões do contêiner do veículo e $\mathbb{L}$ é um conjunto $\{L_1, \ldots, L_k\}$, sendo cada L_v uma lista de caixas para cada vértice $v \in V$, em que o depósito será sempre dado pelo vértice 0. Dado uma instância $I \in \mathcal{I}_{\text{3DTSPL}}$, denotamos por $\mathcal{R}_I$ o conjunto das rotas de I que possuem pelo menos um empacotamento que respeita a ordem em que serão visitados os clientes nessas rotas.

A formulação em programação linear inteira do problema 3DTSPL que consideramos nesse trabalho, é uma adaptação da formulação do problema TSP apresentada na seção 2.1. Nessa formulação foram adicionadas restrições que eliminam as rotas que não possuem um empacotamento que respeite sua ordem. Dada a instância $I = (V, E, c, B, \mathbb{L}) \in \mathcal{I}_{\text{3DTSPL}}$, o problema 3DTSPL pode ser formulado como:

$$\text{minimize} \sum_{e \in E} c_e x_e$$

sujeito a

$$\sum_{e \in \delta(v)} x_e = 2 \qquad \forall v \in V\,, \tag{2.4}$$

$$\sum_{e \in \delta(S)} x_e \geq 2 \qquad \forall S \subset V, \quad S \neq \emptyset\,, \tag{2.5}$$

$$\sum_{e \in R} x_e \leq n - 1 \qquad \forall R \notin \mathcal{R}_I\,, \tag{2.6}$$

$$x_e \in \{0, 1\} \qquad \forall e \in E\,. \tag{2.7}$$

As restrições (2.7) fazem as variáveis x_e, para cada $e \in E$, serem binárias e indicam a pertinência de uma aresta na solução. As restrições (2.4) e (2.5) são as restrições básicas da formulação do TSP e garantem que a solução será um circuito hamiltoniano do grafo de entrada. As restrições em (2.6) restringem o conjunto de rotas apenas para aquelas que podem ter empacotamentos que respeitam a ordem dada.

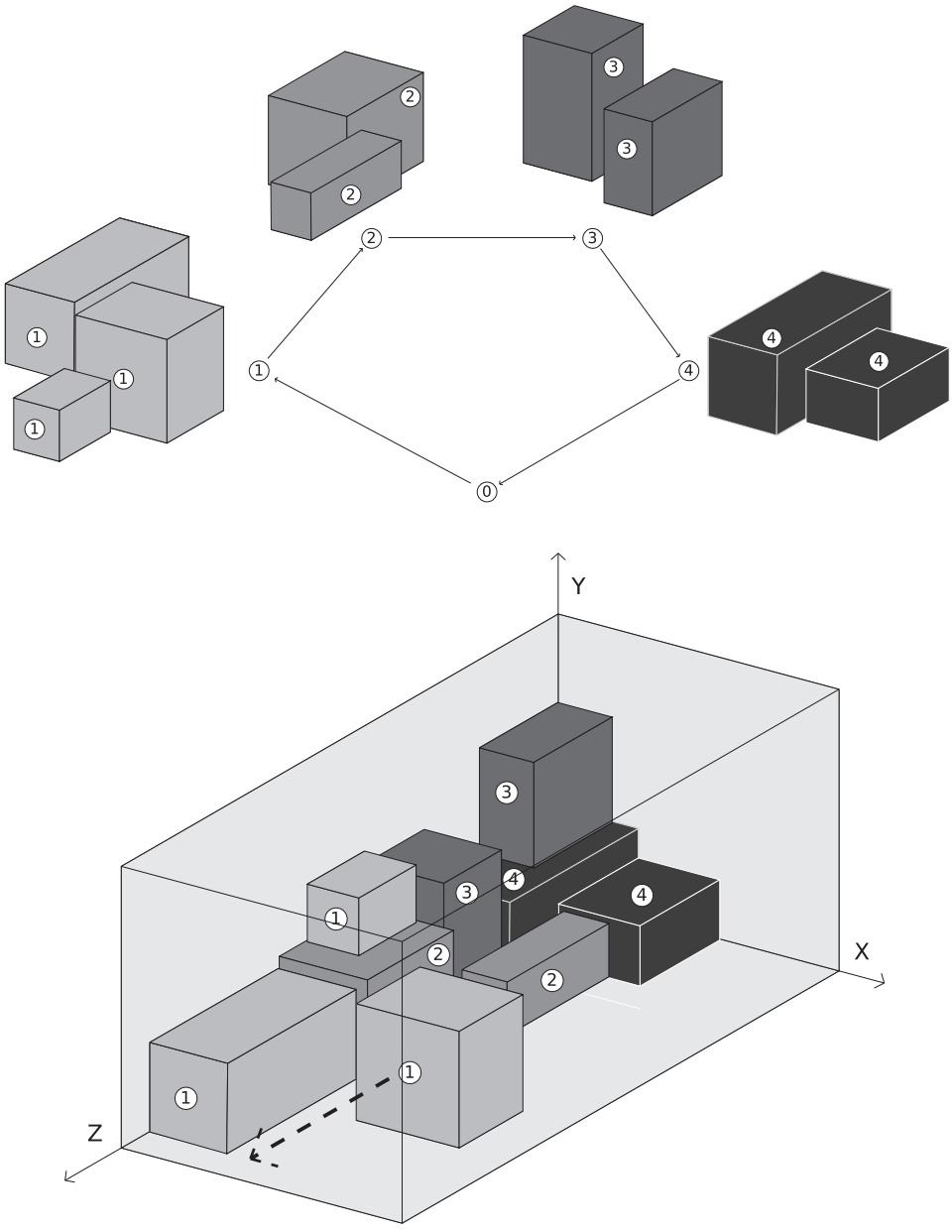


Figura 2.7: Exemplo de rota com os itens de cada cliente e um empacotamento que respeita a ordem da rota.

2.3 Revisão Bibliográfica

O Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional é um problema proposto recentemente, por isso existem poucas publicações sobre ele, mas algumas de suas variantes foram mais estudadas.

Iori, González e Vigo [18] tratam o Problema de Roteamento de Veículos com Restrições de Empacotamento Bidimensional. Nesse problema K veículos idênticos podem ser utilizados para atender os clientes. Os itens são bidimensionais e tem um certo peso, os veículos tem uma superfície bidimensional e uma capacidade máxima. O objetivo é encontrar rotas que atendam todos os clientes, minimizando o total percorrido. Nesse trabalho os autores apresentam um algoritmo *branch-and-cut* para a minimização das rotas e um algoritmo *branch-and-bound* para encontrar empacotamentos viáveis. Heurísticas também são utilizadas para melhorar o desempenho.

Azevedo, Hokama, Miyazawa e Xavier [7] também tratam o Problema de Roteamento de Veículos com Restrições de Empacotamento Bidimensional, apresentando uma abordagem exata para este problema, Também é utilizado um algoritmo *branch-and-cut* para encontrar as rotas. Heurísticas e um algoritmo *branch-and-bound* são utilizados para encontrar um empacotamento viável para cada rota.

Gendreau, Iori, Laporte e Martello [11] tratam o Problema de Roteamento de Veículos com Restrições de Empacotamento Tridimensional, utilizando uma busca Tabu para encontrar rotas que minimizam o custo e chamando uma busca Tabu interna que procura empacotamentos viáveis dos itens tridimensionais em um contêiner. Zachariadis, Tarantilis e Kiranoudis [31] tratam desse mesmo problema, também apresentam uma busca Tabu para resolver o problema de roteamento e uma série de heurísticas para encontrar empacotamentos bidimensionais.

Araújo [5] e Fuellerer, Doerner, Hartl e Iori [9] apresentam meta-heurísticas aplicadas na resolução do Problema de Roteamento de Veículos com Restrições de Empacotamento Tridimensional. Ruan, Ruo, Woghiren e Miao [27] abordam o mesmo problema e apresentam um método que combina algoritmo genético para o roteamento de veículo e busca Tabu para o empacotamento.

Moura e Oliveira [26] abordam o Problema do Roteamento de Veículos com Restrições de Empacotamento Tridimensional e Janelas de Tempo, apresentando heurísticas baseadas em GRASP e busca local.

Cordeau, Iori, Laporte e González [4] apresentaram um algoritmo *branch-and-cut* para o TSP com restrições de Coleta e Entrega com politica de *FIFO* (*First in First out*). Hokama, Azevedo e Miyazawa [17] forneceram um algoritmo *branch-and-cut* para resolver o Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional de forma exata, para o caso em que o empacotamento deve ser realizado por um robô.

Campos [2] apresentou diversas heurísticas para resolver o Problema do Roteamento de

Veículos com Restrições de Empacotamento Tridimensional e Janelas de Tempo.

O Problema de Empacotamento Tridimensional com Restrição de Ordem, no caso onde uma rota pré-definida é dada, também foi pouco estudado. Junqueira, Morabito e Yamashita [21] apresentam uma formulação em Programação Inteira Mista para resolver o empacotamento com ordem acrescido de outras restrições práticas, como o alcance do operador.

Lin, Chang e Yang [23], apresentam uma abordagem baseada em algoritmos genéticos para resolver o Problema de Empacotamento Tridimensional com Restrição de Ordem também acrescido de restrições práticas, como estabilidade. Lai, Xue e Xu [22] tratam do Problema de Empacotamento Tridimensional com Restrição de Ordem apresentando uma modelagem baseada em grafos. Jin, Ohno e Du [20] apresentam algoritmos baseados em meta-heurísticas para esse problema. Armentano e Araújo [1] apresentam uma comparação entre duas classes distintas de restrições de ordem utilizando para tal fim, uma heurística discutida em [6].

Capítulo 3

Técnicas

Este capítulo apresenta algumas técnicas importantes utilizadas na resolução de problemas de otimização. Primeiramente, apresentamos as técnicas de *branch-and-bound* e *branch-and-cut* largamente utilizadas na área de Otimização Combinatória e Pesquisa Operacional. Também apresentamos a técnica de Programação por Restrições que teve seu início na área de Inteligência artificial e está sendo bastante utilizado por outras áreas nas últimas décadas.

3.1 Branch-and-bound

Branch-and-bound é um método bastante utilizado para buscar soluções em diversos problemas de otimização. O método visa uma enumeração das soluções possíveis, baseado no paradigma de "divisão e conquista", procurando manter a eficiência na busca. Utilizando limitantes para o valor de uma solução parcial, este método é capaz de encontrar uma solução ótima sem a necessidade de explorar todo o espaço de busca, isto é, todas as soluções possíveis. Descrevemos de forma geral o funcionamento desse método. Para isso considere o problema a seguir.

Seja E o conjunto de todas as soluções viáveis para o problema

$$\begin{aligned} \max \quad & cx & (3.1)\\ \text{sujeito a} \quad & x \in E. & (3.2) \end{aligned}$$

Agora dividimos o conjunto E em uma coleção finita de subconjuntos menores $E_1, \ldots, E_k$, tal que $E_1 \cup \ldots \cup E_k = E$. Então, resolvemos o seguinte subproblema para cada $i = 1, \ldots, k$,

$$\begin{aligned} \max \quad & cx & (3.3)\\ \text{sujeito a} \quad & x \in E_i. & (3.4) \end{aligned}$$

Seja $z = \max\{cx : x \in E\}$, e $z^i = \max\{cx : x \in E_i\}$ para $i = 1, \ldots, k$. Como $E = E_1 \cup \ldots \cup E_k$, então $z = \max_{i=1,\ldots,k} z^i$, ou seja, escolhendo a melhor solução obtida entre as encontradas nos subproblemas, encontramos a solução ótima para o problema original.

Cada subproblema pode ser resolvido utilizando o mesmo método, dividindo-o em subproblemas e resolvendo os problemas menores. Essa é a parte da divisão (*branch*), e ela pode ser entendida como uma árvore de subproblemas. Na figura 3.1, E foi dividido em E_1 e E_2, depois E_2 foi dividido em E_2' e E_2'', de forma que $E_2' \cup E_2'' = E_2$.

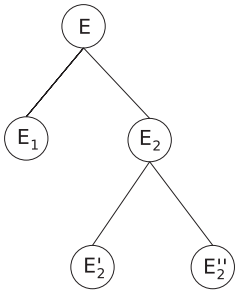


Figura 3.1: Árvore de subproblemas.

Seja $E = E_1 \cup \ldots \cup E_k$ uma decomposição de E em conjuntos menores e $z^i = \max\{cx : x \in E_i\}$ para $i = 1, \ldots, k$, e $\overline{z}(E_i)$ um limitante superior de z^i e $\underline{z}(E_i)$ um limitante inferior de z^i. Então $\overline{z} = \max_{i=1,\ldots,k} \overline{z}(E_i)$ é um limitante superior de z e $\underline{z} = \max_{i=1,\ldots,k} \underline{z}(E_i)$ é um limitante inferior de z.

Além da fase de divisão, também é necessário um algoritmo capaz de computar para todo E_i o limitante inferior $\underline{z}(E_i)$ para o valor de uma solução ótima daquele subproblema, ou seja:

$$\underline{z}(E_i) \leq \max_{x \in E_i} cx.$$

Esse limitante pode ser dado por uma solução viável do problema, obtida de forma heurística.

Da mesma forma, seja $\overline{z}(E_i)$ o limitante superior para a solução ótima de E_i:

$$\overline{z}(E_i) \geq \max_{x \in E_i} cx.$$

Uma forma tradicional de se obter esse limitante superior é utilizar a relaxação linear de uma formulação em programação linear inteira do problema.

Usaremos esses limitantes na fase de poda (*bound*), onde limitamos o espaço de busca do problema. A eficiência dessa parte está diretamente ligada com os limitantes. Quanto mais justos forem, melhor será o desempenho do algoritmo. Infelizmente, nem sempre é fácil encontrar limitantes justos para os problemas de otimização. Descrevemos adiante três tipos de podas. Os dois primeiros exemplos foram retirados de [29].

Poda por Limitante: Podemos limitar o espaço de busca, podando os subproblemas que não podem conter uma solução ótima ou soluções melhores que as já obtidas. A figura 3.2

exemplifica a poda por limitante. Cada E_i da árvore tem indicado o valor de $\overline{z}(E_i)$ na parte superior e $\underline{z}(E_i)$ na parte inferior. Como $20 = \overline{z}(E_1) < \underline{z}(E_2) = 21$ e, se tratando de um problema de maximização, podemos podar o nó E_1, ou seja, não precisamos mais explorar os subproblemas originários de E_1, pois uma solução ótima desse subproblema possui um valor menor que uma solução ótima obtida em E_2

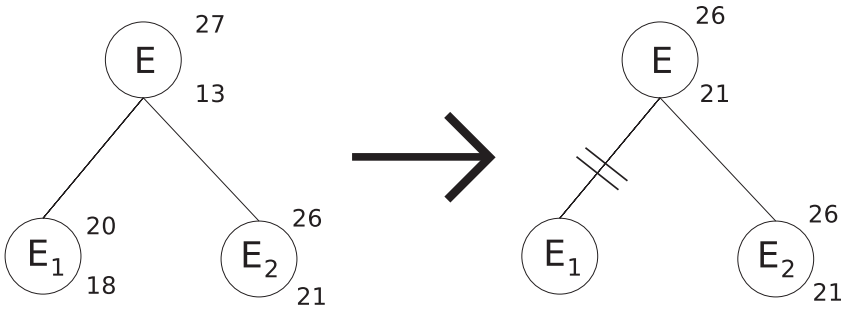


Figura 3.2: Poda por limitante.

Poda por Otimalidade: No exemplo da figura 3.3, considere que $\underline{z}$ é calculado a partir de uma solução viável obtida por uma heurística. Como o limitante superior e o limitante inferior de z^1 são iguais, z^1 = 20, não existe razão para continuar explorando o conjunto E_1, pois a solução heurística encontrada já é ótima para esse subproblema. Então o ramo de E_1 pode ser podado por otimalidade.

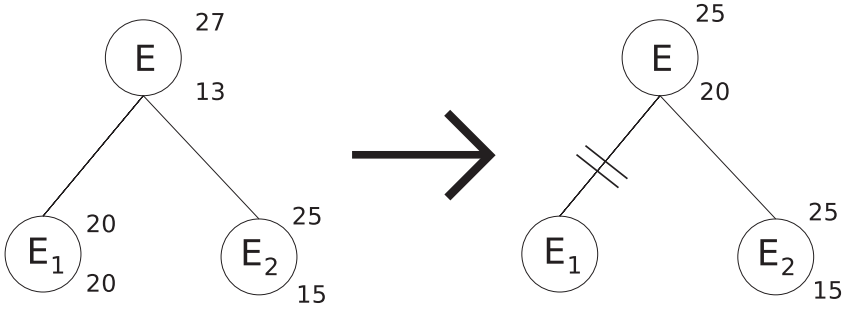


Figura 3.3: Poda por otimalidade.

Poda por Inviabilidade: Podemos também podar ramos que não levem a nenhuma solução viável. Para exemplificar esse tipo de poda, considere o seguinte problema. Temos um conjunto S de 4 itens com pesos 6, 5, 4 e 3. Desejamos escolher um subconjunto S' desses itens com o maior peso possível, desde que não ultrapasse o peso limite $W = 10$. Seja x_i, $i = 1, \ldots, 4$ a variável que indica a pertinência do item i em S' e E é o conjunto de todas as soluções viáveis

para esse problema, ou seja, todos os subconjuntos dos itens 1, 2, 3 e 4, cuja soma dos pesos seja menor ou igual à W. Observando a figura 3.4 considere que vamos subdividir o espaço E de soluções do problema em E_1 e E_2, em que E_1 são os subconjuntos que contêm o item 1 e E_2 os subconjuntos que não contêm o item 1. Da mesma forma subdividimos E_1 em E_1' e E_1'', onde E_1' são os subconjuntos que contêm os itens 1 e 2, e E_1'' são os subconjuntos que contêm o item 1 mas não contêm o item 2. Como os itens 1 e 2 pesam 6 e 5 respectivamente, o peso de qualquer subconjunto contendo esses dois itens ultrapassa W. Portanto $E_1' = \emptyset$, logo podemos podar esse nó.

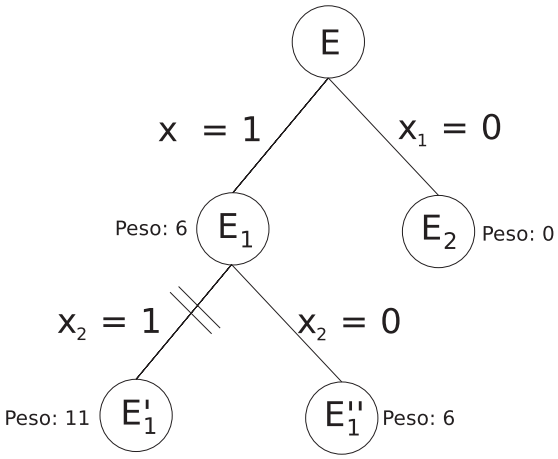


Figura 3.4: Poda por Inviabilidade.

3.2 Planos de Corte

Um plano de corte é uma desigualdade para um programa linear que respeita as restrições do problema, mas permite a remoção de uma solução para uma dada relaxação, que não é solução para o problema original. Para encontrar essas desigualdades é necessário resolver um problema, chamado de *problema da separação*, que depende do problema que está sendo tratado.

O método de planos de corte não se aplica somente a problemas lineares inteiros. Porém esse é o principal foco de nossa atenção. A seguir descrevemos de maneira geral o funcionamento do método de Planos de Cortes. Para isso, consideramos o seguinte programa linear inteiro $\mathcal{Q}$,

$$\begin{aligned} \max \quad & cx \\ \text{sujeito a} \quad & Ax = b \\ & x \geq 0 \\ & x \in \mathbb{Z}, \end{aligned} \tag{3.5}$$

e sua relaxação linear $\mathcal{Q}'$

$$\begin{array}{rl} \max & cx \\ \text{sujeito a} & Ax = b \\ & x \geq 0. \end{array} \tag{3.6}$$

Primeiramente resolvemos a relaxação $\mathcal{Q}'$ e encontramos uma solução ótima x^*. Se x^* é inteira, então também é uma solução ótima para $\mathcal{Q}$. Caso contrário, encontramos uma desigualdade $Dx \geq d$ que satisfaz todas as restrições de $\mathcal{Q}$, mas não satisfaz x^*, ou seja $Dx^* \geq d$ não é válido. Adicionamos $Dx \geq d$ em $\mathcal{Q}'$ e buscamos uma nova solução ótima, garantidamente diferente de x^*. O exemplo seguinte foi retirado de [30].

Exemplo 3.2.1 Planos de Corte
Considere o seguinte programa linear

$$\begin{array}{rl} \min & x_1 - 2x_2 \\ \text{sujeito a} & -4x_1 + 6x_2 \leq 9 \\ & x_1 + x_2 \leq 4 \\ & x_1, x_2 \geq 0 \\ & x_1, x_2 \in \mathbb{Z}, \end{array} \tag{3.7}$$

e sua relaxação linear

$$\begin{array}{rl} \min & x_1 - 2x_2 \\ \text{sujeito a} & -4x_1 + 6x_2 \leq 9 \\ & x_1 + x_2 \leq 4 \\ & x_1, x_2 \geq 0. \end{array} \tag{3.8}$$

A figura 3.5 mostra a região viável da relaxação linear, e sua solução ótima, as soluções viáveis para o problema (3.7) são os pontos dentro dessa região, em que x_1 e x_2 são inteiros.

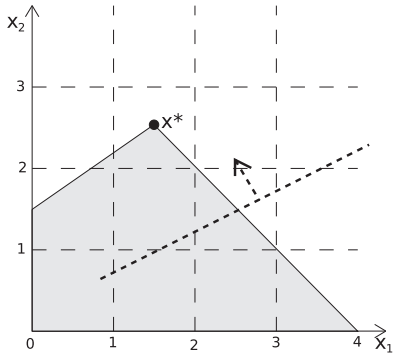


Figura 3.5: Solução ótima para o problema 3.8.

Como a solução encontrada não é inteira, precisamos de uma desigualdade que satisfaça o problema (3.7) mas que elimine a solução encontrada. Uma restrição que atende essas características é $x_2 \leq 2$. Adicionando essa restrição e re-otimizando o problema, encontramos uma nova solução conforme mostra a figura 3.6.

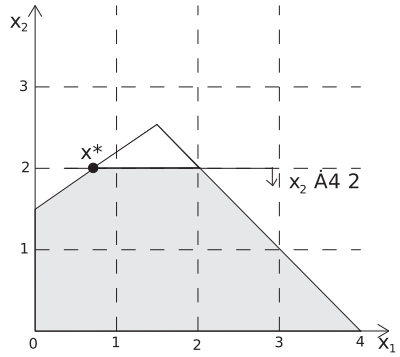


Figura 3.6: Solução ótima com a adição da desigualdade.

Novamente a solução obtida não é inteira, então adicionamos a desigualdade $-3x_1 + 5x_2 \leq 7$ e resolvemos o novo problema. A nova solução encontrada é inteira. Com isso temos uma solução ótima para o problema inteiro (3.7), conforme apresenta a 3.7.

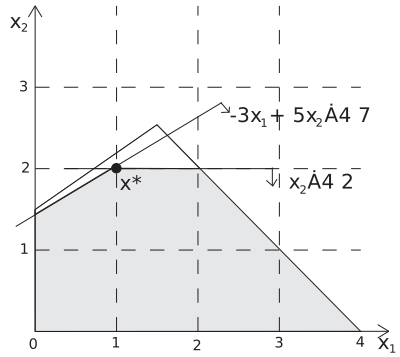


Figura 3.7: Solução ótima inteira.

O método de planos de corte também é utilizado para eliminar soluções inteiras que não são viáveis. Isto pode ser usado quando um problema possui uma classe de desigualdades muito grande e relaxamos o problema retirando essa classe de desigualdades. Considere o seguinte

programa linear inteiro:

$$\begin{array}{rl} \min & cx \\ \text{sujeito a} & Ax \leq b \\ & Dx \leq d \\ & x \in \mathbb{Z}^+, \end{array} \tag{3.9}$$

e a seguinte relaxação:

$$\begin{array}{rl} \min & cx \\ \text{sujeito a} & Ax \leq b \\ & x \geq 0. \end{array} \tag{3.10}$$

Removemos a classe de desigualdades $Dx \leq d$ e a restrição de integralidade. Dessa forma podemos resolver essa relaxação, adicionando cortes que eliminem as soluções inviáveis, da mesma forma que o exemplo 3.2.1. Eliminando também soluções inteiras que violem as desigualdades em $Dx \leq d$.

O método de planos de corte não é restrito a problemas de programação linear inteira, sendo utilizado também em problemas de programação linear e programação inteira mista.

3.2.1 Branch-and-cut

O método *branch-and-cut* é uma técnica para resolver problemas lineares inteiros ou inteiros mistos. Este consiste na adição de planos de corte durante a resolução dos subproblemas encontrados em cada nó da árvore de *branch-and-bound*. Dessa forma há um esforço adicional em cada nó em busca de um limitante dual mais justo.

Essa técnica é interessante quando temos um programa linear inteiro com uma classe de desigualdades grande (exponencial, por exemplo), mas a grande maioria dessas restrições dificilmente será violada, ou não participarão na obtenção da solução ótima. Dessa forma, só as acrescentamos no programa linear inteiro quando tais desigualdades forem violadas.

3.3 Programação por Restrições

Programação por Restrições é um paradigma de programação declarativa que tem se mostrado bastante promissora nas últimas décadas, especialmente para problemas cuja formulação em Programação Linear Inteira é complicada. Essa técnica tem se mostrado bastante eficiente na resolução de grandes problemas combinatórios, em particular em áreas de planejamento e escalonamento. Não somente possui uma forte fundamentação teórica, mas também atrai o interesse comercial prático, particularmente em áreas de modelagem de problemas de otimização e de problemas de satisfação.

Restrições formalizam as relações entre o mundo físico e suas modelagens matemáticas de forma natural e transparente. No contexto de Programação por Restrições, uma restrição

é simplesmente uma relação lógica entre diversas variáveis. A restrição, portanto, restringe os possíveis valores que as variáveis podem assumir simultaneamente, representando uma informação parcial sobre as variáveis do modelo. A principal característica das restrições é sua forma declarativa, ou seja, elas especificam qual relação existe entre as variáveis sem precisar descrever um procedimento computacional para garantir essa relação.

As primeiras ideias de programação por restrições podem ser encontradas na área de Inteligência Artificial nas décadas de sessenta e setenta. A Programação Lógica foi um de seus precursores, sendo que esta e outras formas de programação declarativa utilizam, em geral, a ideia de que o usuário estabelece *o que* deve ser revolvido, e não *como* resolver.

3.3.1 Satisfação de Restrições

Problemas de Satisfação de Restrições (CSP, do inglês Constraint Satisfaction Problem) são definidos da seguinte forma:

- um conjunto de variáveis $X = \{x_1, \ldots, x_n\}$;
- um conjunto finito D_i para cada variável x_i, com os possíveis valores de serem atribuídos a cada variável. Também chamado de domínio de x_i;
- um conjunto de restrições $\mathcal{C}$ que restringe os valores que as variáveis podem assumir simultaneamente.

Os valores, não necessariamente, precisam ser inteiros consecutivos, nem mesmo precisam ser numéricos.

Uma solução para um CSP é uma atribuição de valores de cada domínio para cada uma das respectivas variáveis, de forma que todas as restrições de $\mathcal{C}$ sejam satisfeitas simultaneamente. Na resolução de um CSP podemos encontrar:

- somente uma solução, com nenhuma preferência sobre as outras;
- todas as soluções;
- uma solução boa ou ótima, dada uma certa função objetivo definida sobre as variáveis.

Soluções para um CSP podem ser encontradas explorando sistematicamente as possíveis atribuições de valores às variáveis.

Restrições

As restrições são representadas por uma expressão envolvendo um subconjunto de variáveis. Formalmente, uma restrição $C_{ijk...}$ entre as variáveis x_i, x_j, x_k, ... é qualquer subconjunto das possíveis combinações de valores de x_i, x_j, x_k, ..., isto é, $C_{ijk...} \subseteq D_i \times D_j \times D_k \times ...$. O subconjunto define as combinações de valores que a restrição admite. Na prática as restrições não são representadas dessa forma, mas essa definição enfatiza que as restrições não precisam ser expressões simples. Além disso, elas também não precisam ser lineares. Por exemplo, $x_1 \neq x_2$, $2x_1 = 10x_2 + x_3$ e $x_1x_2 < x_3$.

Técnicas de Busca

A técnica se baseia em uma busca exaustiva, utilizando uma técnica de *backtracking* associada com uma técnica de propagação de restrições, que reduzem significativamente os domínios das variáveis.

Dois níveis de propagação são realizados, a propagação inicial, que vale para toda a árvore do *backtracking*; e, a propagação durante a busca, que elimina temporariamente valores do domínio a cada escolha feita. O domínio é restaurado caso a escolha se mostre inviável, sendo realizada uma nova escolha.

A propagação analisa cada restrição individualmente, removendo do domínio escolhas que não correspondam a atribuições possíveis, ou seja, que violem as restrições. Como a eliminação de alguns valores do domínio de uma variável podem implicar na redução do domínio de outras variáveis, repete-se o processo até que não se consiga mais reduzir qualquer um dos domínios.

Capítulo 4

Algoritmos para o Problema de Empacotamento Tridimensional

Neste capítulo apresentamos alguns métodos para a resolução dos problemas de empacotamento apresentados na seção 2.1.3. Esses métodos serão utilizados como sub-rotina no algoritmo que resolve o 3DTSPL. Dada as rotas encontradas neste último algoritmo, utilizaremos os métodos descritos neste capítulo para encontrar um empacotamento das caixas dos clientes, que respeite a ordem em que serão visitados em cada uma dessas rotas.

4.1 Algoritmo Exato para o Empacotamento por Robô

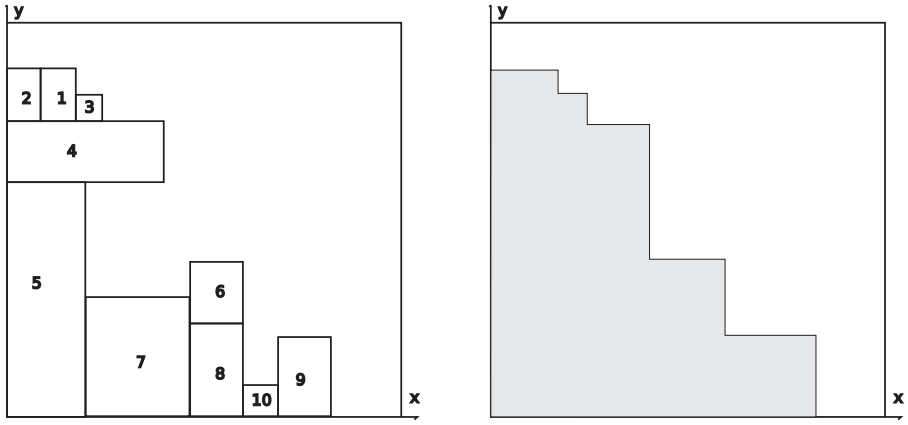
Nesta seção descrevemos um algoritmo proposto por Martello, Pisinger e Vigo [24], para o Problema de Empacotamento por Robô, e uma modificação para que atenda as restrições de ordem. No empacotamento por robô, uma caixa só pode ser colocada a direita, acima ou na frente de outra caixa, e nunca a esquerda, abaixo ou atrás de outra, conforme descrito na seção 2.1. O algoritmo usa a estratégia *branch-and-bound* e enumera o espaço de busca através das posições no contêiner onde uma caixa pode ser colocada sem violar as restrições do empacotamento por robô. Inicialmente apresentamos alguns conceitos necessários para descrever o algoritmo.

Para encontrar as posições onde um item pode ser alocado, definimos a região já ocupada ou bloqueada pelo empacotamento. Essa região recebe o nome de *envelope*. O envelope de um empacotamento é toda a região ocupada pelo empacotamento e toda região atrás, a esquerda e abaixo dos itens. Para formalizar o conceito do envelope de um empacotamento tridimensional, precisaremos conhecer o mesmo conceito para o empacotamento bidimensional. No caso bidimensional, podemos definir o envelope como a área ocupada por um empacotamento e toda a área a esquerda e abaixo de cada item. Formalmente podemos definir o envelope da forma como segue.

Definição 4.1.1. Seja L um conjunto de itens bidimensionais, cada item $i \in L$ com dimensões (w_i, h_i), e $\mathcal{Q}$ um empacotamento desses itens, que atribui uma posição (x_i, y_i) a cada item. Em um recipiente bidimensional $\hat{B} = (W, H)$. Um envelope de $\mathcal{Q}$ é a área definida por

$$S(\mathcal{Q}) = \{(x, y) \in \mathbb{R}^2_+ : \exists i \in L \text{ com } x < x_i + w_i \text{ e } y < y_i + h_i\}.$$

A figura 4.1a mostra um exemplo de empacotamento bidimensional de dez itens e a figura 4.1b mostra seu respectivo envelope. Note que toda a área abaixo e a esquerda dos itens empacotados também pertencem ao envelope. Definimos o complemento $\bar{S}(\mathcal{Q})$ de $S(\mathcal{Q})$ como a área de $\hat{B}$ que não pertence ao envelope, ou seja, $\bar{S}(\mathcal{Q}) = [0, W) \times [0, H) - S(\mathcal{Q})$.



(a) Empacotamento (b) Envelope

Figura 4.1: (a) Um exemplo de empacotamento bidimensional e (b) seu respectivo envelope.

Uma forma de delimitar um envelope é a utilização dos *pontos de canto*, que são os pontos de $\bar{S}(\mathcal{Q})$ mais abaixo e a esquerda. Estes pontos delimitam as bordas horizontais e verticais do envelope, formalmente, definimos um conjunto de pontos de canto $\hat{C}(\mathcal{Q})$ da maneira como segue.

Definição 4.1.2. Dado um empacotamento $\mathcal{Q}$, o conjunto de pontos de canto $\hat{C}(\mathcal{Q})$ é definido por,

$$\hat{C}(\mathcal{Q}) = \{(x, y) \in \bar{S}(\mathcal{Q}) : \nexists (x', y') \in \bar{S}(\mathcal{Q}) \backslash \{(x, y)\}, x' \leq x \text{ e } y' \leq y\}.$$

A figura 4.2 exemplifica os pontos de canto do empacotamento da figura 4.1a, que são indicados pelos pontos pretos.

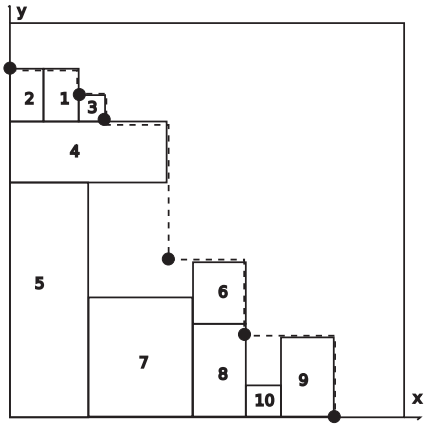


Figura 4.2: Exemplo de pontos de canto (2D) [24].

Descrevemos no algoritmo 1 a rotina 2D-Corners, apresentado em [24], que encontra os pontos de canto de um empacotamento bidimensional qualquer. Para isso, considere $\mathcal{Q}$ um empacotamento de um conjunto de itens L, que atribui uma posição (x_i, y_i) para cada item i em L, e J o conjunto dos itens ainda não empacotados. Consideramos que o algoritmo recebe os itens ordenados pelos *pontos extremos*, que são seus cantos superiores direitos (x_j+w_j, y_j+h_j), de forma que os valores de $y_j + h_j$ são não-crescentes. Desempatamos pelo maior valor de $x_j + w_j$. No exemplo da figura 4.2 o número no centro dos itens indica a posição de cada item nessa ordenação.

Nas linhas 4-8, o algoritmo 2D-Corners atribui em e_i, $i = 1, \ldots, m$ todos os itens que definem as bordas verticais do envelope. No exemplo da figura 4.2 são os itens 1, 3, 4, 6 e 9, nessa ordem. A variável auxiliar $\bar{x}$ guarda o valor da última borda encontrada. Então, nas linhas 10-12, para cada item e_i, $i = 2 \ldots m$, o algoritmo adiciona ao conjunto de pontos de canto $\hat{C}$ o ponto $(x_{e_{j-1}} + w_{e_{j-1}}, y_{e_j} + h_{e_j})$. No exemplo da figura 4.2 os pontos adicionados são $(x_1 + w_1, y_3 + h_3), (x_3 + w_3, y_4 + h_4), (x_4 + w_4, y_6 + h_6)$ e $(x_6 + w_6, y_9 + h_9)$. Também são adicionados os pontos $(0, y_1 + h_1)$ e $(x_9 + w_9, 0)$, nas linhas 9 e 13 respectivamente. Por fim, nas linhas 14-18 são removidos os pontos onde nenhum dos itens de J pode ser alocado, pois ultrapassa o limite de $\hat{B}$. Finalmente o algoritmo devolve $\hat{C}$.

A ordenação por pontos extremos não é realizada dentro do algoritmo, pois o 2D-Corners será utilizado diversas vezes como sub-rotina para o algoritmo 3D-Corners, que descrevermos mais tarde, e essa ordenação só será necessária uma vez por cada chamada de 3D-Corners.

Algoritmo 1: 2D-Corners

Entrada: Uma lista de itens L ordenados pelos seus pontos extremos; um empacotamento $\mathcal{Q}$ desses itens em $\hat{B}$; que atribui uma posição (x_i, y_i) para cada item $i \in L$; e, o conjunto dos itens não empacotados J.

Saída: Um conjunto $\hat{C}(\mathcal{Q})$ de pontos de canto;

1 **se** $L = \emptyset$ **então devolva** $\{(0,0)\}$;
2 $\hat{C} = \emptyset$;
3 $\bar{x} := m := 0$;
4 **para** $j := 1$ ***to*** $|L|$ **faça**
5 **se** $x_j + w_j > \bar{x}$ **então**
6 $m := m + 1$; $e_m := j$; $\bar{x} := x_j + w_j$;
7 **fim**
8 **fim**
9 $\hat{C} := \{(0, y_{e_1} + h_{e_1})\}$;
10 **para** $j := 2$ ***até*** m **faça**
11 $\hat{C} := \hat{C} \cup \left\{(x_{e_{j-1}} + w_{e_{j-1}}, y_{e_j} + h_{e_j})\right\}$;
12 **fim**
13 $\hat{C} := \hat{C} \cup \{(x_{e_m} + w_{e_m}, 0)\}$;
14 **para cada** $(x'_j, y'_j) \in \hat{C}$ **faça**
15 **se** $x'_j + \min\{w_i : i \in J\} > W$ ***ou*** $y'_j + \min\{h_i : i \in J\} > H$ **então**
16 $\hat{C} := \hat{C} \backslash \left\{(x'_j, y'_j)\right\}$;
17 **fim**
18 **fim**
19 **devolva** $\hat{C}$.

O algoritmo 2D-Corners encontra corretamente os pontos de canto, de um empacotamento bidimensional. Para mostrar isso, considere um empacotamento $\mathcal{Q}$ dos itens de L em $\hat{B}$, e (x_i, y_i) a posição atribuída por $\mathcal{Q}$ para cada item i em L. Também considere e_j, $j = 1, \ldots m$, os itens encontrados nas linhas 4-8 do algoritmo 2D-Corners. Primeiramente precisaremos do seguinte lema.

Lema 4.1.1. No fim do laço das linhas 4-8, para todo $j = 2, \ldots, m$,

$$y_{e_j} + h_{e_j} < y_{e_{j-1}} + h_{e_{j-1}} \text{ e}$$

$$x_{e_{j-1}} + w_{e_{j-1}} < x_{e_j} + w_{e_j}.$$

Demonstração. Como os itens são ordenados de forma não-crescente por $y_i + h_i$, desempatando por $x_i + w_i$ decrescente, e considerados somente nessa ordem, é evidente que $y_{e_{j-1}} + h_{e_{j-1}} \geq$

$y_{e_j} + h_{e_j}$. Porém, se dois itens tem a borda $y_i + h_i$ no mesmo valor, somente o mais a esquerda será considerado. Portando $y_{e_{j-1}} + h_{e_{j-1}} \neq y_{e_j} + h_{e_j}$. Logo vale que $y_{e_j} + h_{e_j} < y_{e_{j-1}} + h_{e_{j-1}}$.

Para mostrar a segunda parte do lema, verificamos que os elementos são atribuídos em e_j somente se aumentarem o valor de $\bar{x}$ na linha 5 do algoritmo, logo $x_{e_j} + w_{e_j} > x_{e_{j-1}} + w_{e_{j-1}}$. □

Teorema 4.1. Os pontos $(x_{e_{j-1}} + w_{e_{j-1}}, y_{e_j} + h_{e_j})$, para $j = 2, \ldots, m$, são pontos de cantos do empacotamento $\mathcal{Q}$.

Demonstração. Para que $(x_{e_{j-1}} + w_{e_{j-1}}, y_{e_j} + h_{e_j})$ seja um ponto de canto, segundo a definição 4.1.2, duas propriedades devem ser atendidas:

1. o ponto $(x_{e_{j-1}} + w_{e_{j-1}}, y_{e_j} + h_{e_j})$ deve pertencer à $\bar{S}(\mathcal{Q})$, e
2. não deve existir (x', y') diferente de $(x_{e_{j-1}} + w_{e_{j-1}}, y_{e_j} + h_{e_j})$ em $\bar{S}(\mathcal{Q})$ tal que $x' \leq x_{e_{j-1}} + w_{e_{j-1}}$ e $y' \leq y_{e_j} + h_{e_j}$.

É fácil ver que a propriedade (1) vale. Para mostrar que a propriedade (2) vale, suponha por absurdo que exista (x', y') em $\bar{S}(\mathcal{Q})$, com $(x', y') \neq (x_{e_{j-1}} + w_{e_{j-1}}, y_{e_j} + h_{e_j})$ que fere a propriedade (2), ou seja, $x' \leq x_{e_{j-1}} + w_{e_{j-1}}$ e $y' \leq y_{e_j} + h_{e_j}$. Como (x', y') deve ser diferente de $(x_{e_{j-1}} + w_{e_{j-1}}, y_{e_j} + h_{e_j})$, x' e y' não podem ser simultaneamente iguais a $x_{e_{j-1}} + w_{e_{j-1}}$ e $y_{e_j} + h_{e_j}$. Então somente dois casos podem acontecer:

caso 1:

$$x' \leq x_{e_{j-1}} + w_{e_{j-1}} \text{ e } y' < y_{e_j} + h_{e_j}.$$

Nesse caso, usando o lema 4.1.1 temos que $x_{e_{j-1}} + w_{e_{j-1}} < x_{e_j} + w_{e_j}$, portanto

$$\begin{aligned} x' \leq x_{e_{j-1}} + w_{e_{j-1}} < x_{e_j} + w_{e_j} \text{ e } y' < y_{e_j} + h_{e_j}, \\ x' < x_{e_j} + w_{e_j} \text{ e } y' < y_{e_j} + h_{e_j}, \end{aligned}$$

mas pela definição 4.1.1 de envelope, $(x', y') \in S(\mathcal{C})$, pois o item e_j está acima e a direita desse ponto. Logo $(x', y') \notin \bar{S}(\mathcal{C})$, o que contradiz a hipótese.

Caso 2:

$$x' < x_{e_{j-1}} + w_{e_{j-1}} \text{ e } y' \leq y_{e_j} + h_{e_j}.$$

Nesse caso, também usando o lema 4.1.1 temos que $y_{e_j} + h_{e_j} < y_{e_{j-1}} + h_{e_{j-1}}$, portanto podemos afirmar que

$$\begin{aligned} x' < x_{e_{j-1}} + w_{e_{j-1}} \text{ e } y' \leq y_{e_j} + h_{e_j} < y_{e_{j-1}} + h_{e_{j-1}}, \\ x' < x_{e_{j-1}} + w_{e_{j-1}} \text{ e } y' < y_{e_{j-1}} + h_{e_{j-1}}. \end{aligned}$$

Novamente, pela definição 4.1.1 de envelope, $(x', y') \in S(\mathcal{C})$, pois o item e_{j-1}, está acima e a direita de (x', y'). Logo $(x', y') \notin \bar{S}(\mathcal{C})$, o que também contradiz a hipótese.

Portando, tal (x', y') não pode existir em $\bar{S}(\mathcal{C})$, de modo que $(x_{e_{j-1}} + w_{e_{j-1}}, y_{e_j} + h_{e_j})$ é um ponto de canto.

□

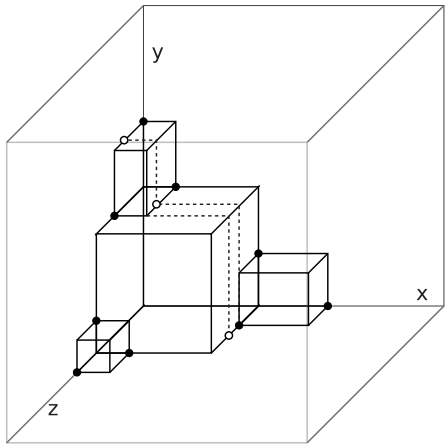


Figura 4.3: Exemplo de pontos de canto (3D).

Como $x = 0, x_{e_1} + w_{e_1}, \ldots, x_{e_m} + w_{e_m}$ definem todas as bordas verticais do envelope, o algoritmo 2D-Corners encontra todos os pontos de canto do empacotamento, exceto os que são removidos por inviabilidade.

Lema 4.1.2. Seja L o conjunto de caixas empacotadas, J o conjunto de caixas não empacotadas, e n o número de caixas em $L \cup J$, a complexidade computacional do algoritmo 2D-Corners é $O(n)$.

Demonstração. O laço das linhas 4-8 itera por $|L|$ vezes. O laço das linhas 10-12 tem complexidade de tempo $O(|L|)$. As linhas 15-17 tem complexidade $O(1)$, pois o mínimo só precisa ser encontrado uma vez, o que pode ser feito em $O(|J|)$, e o laço das linhas 14-18 tem complexidade de tempo $O(|L|)$. Portanto a complexidade do 2D-Corners é $O(|L|) + O(|J|) = O(n)$. □

No caso tridimensional usaremos o conceito de pontos de canto para encontrar as posições onde um próximo item pode ser alocado. Primeiramente definimos o envelope para o caso tridimensional.

Definição 4.1.3. Considere I um conjunto de itens tridimensionais e $\mathcal{P}$ um empacotamento desses itens em um recipiente tridimensional $B = (W, H, D)$. O envelope de $\mathcal{P}$ é uma região definida por:

$$S(\mathcal{P}) = \{(x, y, z) \in \mathbb{R}^3_+ : \exists i \in I \text{ com } x < x_i + w_i, y < y_i + h_i \text{ e } z < z_i + d_i\}.$$

O complemento do envelope $S(\mathcal{P})$ é a região $\bar{S}(\mathcal{P}) = [0, W) \times [0, H) \times [0, D) - S(\mathcal{P})$.

Definição 4.1.4. Os pontos de canto $C(\mathcal{P})$ de um empacotamento $\mathcal{P}$ são os pontos mais abaixo, a esquerda e atrás de $\bar{S}(\mathcal{P})$. Formalmente,

$$C(\mathcal{P}) = \{(x, y, z) \in \bar{S}(\mathcal{P}) : \nexists (x', y', z') \in \bar{S}(\mathcal{P}) \backslash \{(x, y, z)\}, x' \leq x, y' \leq y \text{ e } z' \leq z\}.$$

A figura 4.3, mostra um exemplo de empacotamento tridimensional. Os pontos pretos representam os pontos de canto.

O algoritmo 2 descreve a rotina 3D-Corners, apresentada por Martello et al. [24], que encontra o conjunto $C(\mathcal{P})$ de pontos de canto do empacotamento tridimensional $\mathcal{P}$. Inicialmente, na linha 2, o algoritmo gera um conjunto $\mathcal{Z} := \{0\} \cup \{z_i + d_i : i \in I\}$ de todas as posições z onde termina um item e também de $z = 0$. Na linha 3 o algoritmo ordena $\mathcal{Z}$ de forma crescente, tal que $\mathcal{Z} = \{z'_1, \ldots, z'_r\}$.

Na linha 4, o algoritmo gera o conjunto de pontos de canto C, inicialmente vazio, e uma variável auxiliar k. No laço das linhas 5-15, para cada z'_k do conjunto $\mathcal{Z}$ são gerados os conjuntos $I_k := \{i \in I : z_i + d_i > z'_k\}$. Então, é aplicado o algoritmo 2D-Corners sobre o empacotamento dos itens de I_k, obtendo o conjunto de pontos de canto bidimensionais $\hat{C}_k$. Note que nesse passo o algoritmo 2D-Corners vai ignorar a posição z de cada item, fazendo surgir sobreposições de itens. Porém, isso não afetará o comportamento do algoritmo.

Para cada ponto $(x'_j, y'_j) \in \hat{C}_k$ são adicionados os pontos (x'_j, y'_j, z'_k) em C caso esses não sejam dominados, ou seja, se $(x'_j, y'_j) \notin \hat{C}_{k-1}$. Por fim, o algoritmo devolve C. A figura 4.3 mostra o conjunto de pontos de canto $C(\mathcal{P})$ encontrado aplicando-se o algoritmo 2D-Corners cinco vezes, uma para cada valor de z onde termina um item no eixo Z e para $z = 0$. Pontos de cantos dominados, chamados de *falsos pontos de canto*, estão representados por círculos vazios.

Algoritmo 2: 3D-Corners

Entrada: Um conjunto dos itens I empacotados, um empacotamento $\mathcal{P}$, e um conjunto dos itens não empacotados J.

Saída: Um conjunto $C(\mathcal{P})$ de pontos de canto;

1 **se** $I = \emptyset$ **então devolva** $\{(0, 0, 0)\}$;
2 $\mathcal{Z} := \{0\} \cup \{z_i + d_i : i \in I\}$; //Não duplicar valores em $\mathcal{Z}$;
3 ordenar $\mathcal{Z}$ de forma crescente, e seja $\mathcal{Z} = \{z'_1, \ldots, z'_r\}$;
4 $C := \hat{C}_0 := \emptyset$; $k := 1$;
5 **enquanto** $k \leq r$ ***e*** $z'_k + min\{d_i : i \in J\} \leq D$ **faça**
6 $I_k := \{i \in I : z_i + d_i > z'_k\}$;
7 aplicar 2D-CORNERS sobre o empacotamento de I_k obtendo $\hat{C}_k$;
8 **para cada** $(x'_j, y'_j) \in \hat{C}_k$ **faça**
9 //Exclui falsos pontos de canto;
10 **se** $(x'_j, y'_j) \notin \hat{C}(I_{k-1})$ **então**
11 $C := C \cup (x'_j, y'_j, z'_k)$;
12 **fim**
13 **fim**
14 $k* := k$; $k := k + 1$;
15 **fim**
16 **devolva** C;

Para demonstrar a corretude do algoritmo, considere o conjunto de itens I e o empacotamento

tridimensional $\mathcal{P}$ que define posições (x_i, y_i, z_i) para cada item $i \in I$.

Lema 4.1.3. Os pontos (x'_j, y'_j, z'_k) encontrados na linha 11 do algoritmo 3D-Corners, são pontos de canto de $\mathcal{P}$.

Demonstração. Pela definição 4.1.4, o ponto (x'_j, y'_j, z'_k) é ponto de canto, se atende a duas propriedades,

1. (x'_j, y'_j, z'_k) pertence a $\bar{S}(\mathcal{P})$, e
2. não existe (x', y', z') diferente de (x'_j, y'_j, z'_k), em $\bar{S}(\mathcal{P})$ tal que $x' \leq x'_j$, $y' \leq y'_j$ e $z' \leq z'_k$.

É fácil ver que a propriedade (1) vale. Para mostrar que a propriedade (2) vale, suponha por absurdo que exista (x', y', z') que fere a propriedade (2), ou seja $(x', y', z') \in \bar{S}(\mathcal{P})$ e $x' \leq x'_j$, $y' \leq y'_j$ e $z' \leq z'_k$. Como $(x', y', z') \neq (x'_j, y'_j, z'_k)$, três casos podem existir.

Caso 1: $x' < x'_j$, $y' \leq y'_j$ e $z' \leq z'_k$ e *Caso 2:* $x' \leq x'_j$, $y' < y'_j$ e $z' \leq z'_k$. Se caso 1 ou caso 2 ocorrer, $(x', y', z') \in S(\mathcal{P})$, pois (x'_j, y'_j) define o envelope bidimensional em z'_k. Toda região atrás dessa região pertence à $S(\mathcal{P})$, contradizendo a hipótese de que $(x', y', z') \in \bar{S}(\mathcal{P})$.

Caso 3:

$$x' \leq x'_j, y' \leq y'_j \text{ e } z' < z'_k,$$

Como o ponto (x'_j, y'_j, z'_k) foi adicionado na linha 11 do algoritmo 3D-Corners, pela condição da linha 10, sabemos que $(x'_j, y'_j) \notin C(I_{k-1})$. Como $I_k \subset I_{k-1}$, e (x'_j, y'_j) não é um ponto de canto de $C(I_{k-1})$, existe algum item i em I_{k-1}, tal que $z_i + d_i = z'_k$, $x'_j < x_i + w_i$ e $y'_j < y_i + h_h$. Como $x' < x_i + w_i, y' < y_i + h_i$ e $z' < z_i + d_i$, pela definição 4.1.3, o ponto $(x'_j, y'_j, z'_k) \in S(\mathcal{P})$. O que contradiz a hipótese de que $(x', y', z') \in \bar{S}(\mathcal{P})$.

Portanto, tal ponto (x', y', z') não pode existir e (x'_j, y'_j, z'_k) é um ponto de canto. □

Dessa forma, como $z = 0, z'_1, \ldots, z'_r$, define todas as bordas do envelope de $\mathcal{P}$ no eixo Z, o algoritmo encontra todos os pontos de canto.

Lema 4.1.4. Seja n o número de caixas em I, a complexidade computacional do algoritmo 3D-Corners é $O(n^2)$.

Demonstração. Há no máximo $|I| + 1$ coordenadas distintas em $\mathcal{Z}$. Cada conjunto $\hat{C}_k$ é encontrado por 2D-Corners em tempo $O(n)$, pois a ordenação dos itens pode ser feita apenas uma vez no início do algoritmo 3D-Corners em $O(n \log n)$. Para cada valor z'_k, a verificação dos pontos de canto na linha 10 antes da adição deles em C requer tempo $O(|\hat{C}_{k-1}|)$ no total, pois tanto $\hat{C}_{k-1}$ como $\hat{C}_k$ estão ordenados de forma crescente em x'_j e decrescente em y'_j. Cada iteração do laço 5-15 executa em tempo $O(n)$ por no máximo $n + 1$ iterações. A ordenação da linha 3 pode ser feita em tempo $O(n \log n)$. Logo, a complexidade computacional deste algoritmo é $O(n^2)$. □

Dessa forma temos um método para encontrar as posições onde inserir um item, sem violar a restrição de empacotamento por robô. O algoritmo exato, denominado por *OneBin_Robot*, para o Problema de Empacotamento por Robô, apresentado em [24] usa a abordagem *branch-and-bound* explorando os pontos de canto do empacotamento. As caixas são consideradas em uma ordem específica, definida pela entrada, e a busca é feita de maneira recursiva. No início, nenhuma caixa está empacotada e portanto $C(\mathcal{P}) = \{(0,0,0)\}$. A cada chamada da rotina, são computados os pontos de canto e cada item não empacotado é atribuído um a um, para cada ponto de canto encontrado por 3D-Corners. Então OneBin_Robot é chamado recursivamente. Quando um item não puder ser empacotado em nenhum dos pontos de canto, faz-se um *backtrack*.

Resta ainda mostrar que o algoritmo que utiliza os pontos de canto encontra corretamente um empacotamento por robô. O teorema 4.2 mostra que qualquer empacotamento por robô possui um equivalente que pode ser encontrado pelo algoritmo OneBin_Robot.

Teorema 4.2. Qualquer empacotamento por robô possui um empacotamento equivalente, sendo cada item posicionado em um ponto de canto.

Demonstração. Conforme definimos, um empacotamento por robô é aquele que pode ser obtido empacotando as caixas de forma que cada caixa somente pode ser posicionada acima, a direita, ou a frente das caixas já alocadas. Isto é equivalente a dizer que nenhuma caixa alocada intersecciona o envelope do empacotamento anterior. Dessa forma, considere um empacotamento qualquer $\mathcal{P}'$ por robô e uma ordenação $(l_1, l_2, \ldots, l_n)$ das caixas de L, tal que as caixas podem ser colocadas no contêiner nessa ordem de forma que cada caixa seja posicionada fora do envelope formado pelo empacotamento anterior.

Vamos mostrar que existe um empacotamento $\mathcal{P}$, em que as caixas podem ser colocadas na ordem $(l_1, l_2, \ldots, l_n)$, sempre em pontos de canto. Considere $\mathcal{P}'_i$ e $\mathcal{P}_i$ os empacotamentos parciais de $\mathcal{P}'$ e $\mathcal{P}$ composto pelas caixas $(l_1, \ldots, l_i)$. Além disso, seja (x'_i, y'_i, z'_i) a posição do item i no empacotamento $\mathcal{P}'$ e (x_i, y_i, z_i) a posição do item i no empacotamento $\mathcal{P}$.

Para mostrar que o empacotamento $\mathcal{P}$ é equivalente ao empacotamento $\mathcal{P}'$, vamos mostrar que para todo empacotamento parcial $\mathcal{P}'_i$ e $\mathcal{P}_i$, $S(\mathcal{P}_i) \subseteq S(\mathcal{P}'_i)$, o que é equivalente à $\bar{S}(\mathcal{P}'_i) \subseteq \bar{S}(\mathcal{P}_i)$, ou seja a região livre (complemento do envelope) no empacotamento $\mathcal{P}_i$ contêm toda a região livre do empacotamento $\mathcal{P}'_i$. Isto é verdade para quando todo i, $x'_i \leq x_i$, $y'_i \leq y_i$ e $z'_i \leq z_i$. Para mostrar que essa propriedade vale, vamos utilizar indução em i.

Como o primeiro item em $\mathcal{P}$ será colocado no ponto $(0,0,0)$, pois esse é o único ponto de canto disponível, é evidente que vale $x'_1 \leq x_1$, $y'_1 \leq y_1$ e $z'_1 \leq z_1$.

Para cada $i = 2 \ldots n$ é possível encontrar um ponto de canto (x_i, y_i, z_i) em $S(\mathcal{P}_{i-1})$, que atenda a propriedade $x'_i \leq x_i$, $y'_i \leq y_i$ e $z'_i \leq z_i$.

Como (x'_i, y'_i, z'_i) pertence a $\bar{S}(\mathcal{P}'_{i-1})$ e $\bar{S}(\mathcal{P}'_{i-1}) \subseteq \bar{S}(\mathcal{P}_{i-1})$, (x'_i, y'_i, z'_i) pertence a $\bar{S}(\mathcal{P}_{i-1})$. Portanto, podemos encontrar um $\bar{x} \leq x'_i$ tal que $(\bar{x}, y'_i, z'_i) \in \bar{S}(\mathcal{P}_{i-1})$, e não existe $\bar{x}' < \bar{x}$ tal

que $(\bar{x}', y'_i, z'_i) \in \bar{S}(\mathcal{P}_{i-1})$. Intuitivamente, $(\bar{x}, y'_i, z'_i)$ seria um ponto atrás de (x'_i, y'_i, z'_i) (ou o próprio), que "encosta" no envelope de $\mathcal{P}_{i-1}$.

Então, encontramos um $\bar{y} \leq y'_i$ tal que $(\bar{x}, \bar{y}, z'_i) \in \bar{S}(\mathcal{P}_{i-1})$ e não existe $\bar{y}' < \bar{y}$ tal que $(\bar{x}, \bar{y}', z'_i) \in \bar{S}(\mathcal{P}_{i-1})$. Finalmente, encontramos um $\bar{z} \leq z'_i$ tal que $(\bar{x}, \bar{y}, \bar{z}) \in \bar{S}(\mathcal{P}_{i-1})$ e não existe $\bar{z}' < \bar{z}$ tal que $(\bar{x}, \bar{y}, \bar{z}') \in \bar{S}(\mathcal{P}_{i-1})$.

Pela forma que encontramos o ponto $(\bar{x}, \bar{y}, \bar{z})$ garantimos que este é um ponto de canto, pois não existe $(\bar{x}', \bar{y}', \bar{z}')$ diferente de $(\bar{x}, \bar{y}, \bar{z})$ em $\bar{S}(\mathcal{P}_{i-1})$, tal que $\bar{x}' \leq \bar{x}$, $\bar{y}' \leq \bar{y}$ e $\bar{z}' \leq \bar{z}$. Assim, consideramos $(x_i, y_i, z_i) = (\bar{x}, \bar{y}, \bar{z})$.

□

Adaptamos esse algoritmo de forma que sempre que uma caixa é atribuída a um ponto verificamos se ela bloqueia outra já empacotada. Se isto acontecer, também é feito um *backtrack*. Pelos experimentos realizados, optamos pela ordenação das caixas na ordem inversa com que serão descarregadas. Dessa forma, as caixas que serão descarregadas por último serão as primeiras a serem empacotadas. O desempate é feito pela caixa de maior volume, uma vez que, no geral, obteve um melhor desempenho do algoritmo OneBin_Robot.

Para agilizar o processo de enumeração, o ramo de uma chamada com empacotamento $\mathcal{P}$, contendo uma lista de caixas I, e um conjunto de caixas J ainda não empacotados, pode ser podado caso não seja considerado promissor, segundo o Lema 4.1.5.

Lema 4.1.5. Seja J o conjunto de itens ainda não empacotados e $\mathcal{P}$ o empacotamento do conjunto de itens I em B, V_B o volume de B, $V_\mathcal{P}$ o volume do envelope $S(\mathcal{P})$, e V_J a soma do volume de cada caixa em J. Se $V_B - V_\mathcal{P} < V_J$, então não é possível completar o empacotamento dos itens de J.

Demonstração. Os próximos itens só são alocados no complemento de $S(\mathcal{P})$ e o volume de $\bar{S}(\mathcal{P})$ é dado por $V_B - V_\mathcal{P}$. Se o volume de $\bar{S}(\mathcal{P})$ é menor que o volume das caixas em J, temos que não existe um empacotamento para J em $\bar{S}(\mathcal{P})$. □

Pelo Lema 4.1.5 o algoritmo poda o nó caso $V_B - V_\mathcal{P} < V_J$. O algoritmo exato também pode funcionar como uma heurística no algoritmo para o 3DTSPL, restringindo-se o tempo de sua execução. Denominamos a adaptação do algoritmo OneBin_Robot, que considera restrições de ordem, por MPV na sua forma exata e por MPV*T*, a heurística obtida limitando o algoritmo MPV a uma execução de T segundos.

4.2 Algoritmo Exato baseado em programação por restrições

Nesta seção vamos descrever um algoritmo exato para o empacotamento tridimensional, apresentado por Martello, Pisinger, Vigo, Boef e Korst [25], e uma modificação desse para considerar as restrições de ordem. Diferentemente do empacotamento por robô, não podemos utilizar

a mesma estratégia do algoritmo OneBin_Robot, que consiste em posicionar sucessivamente as caixas nos pontos de canto. Ao invés disso, uma solução baseada em programação por restrições foi apresentada.

Em um empacotamento, todas as caixas são colocadas dentro do contêiner e duas caixas não podem se sobrepor. O último critério é satisfeito se para cada par de caixas i, j, a caixa i está à esquerda, direita, abaixo, acima, atrás ou na frente da caixa j. Para modelar essa relação, denotamos para cada par de caixas i, j, uma variável r_{ij}. O domínio de cada uma dessas variáveis será definido por um conjunto de posição relativas $Dom(r_{ij}) = \{\texttt{l}, \texttt{r}, \texttt{u}, \texttt{a}, \texttt{b}, \texttt{f}\}$ (respectivamente *left of*, *right of*, *under*, *above*, *behind*, ou *in front of*), representando assim cada uma das possíveis relações entre as caixas i e j.

O problema é resolvido recursivamente pelo algoritmo *OneBin_General*. A cada chamada, duas caixas i e j são consideradas e uma das posições relativas em $Dom(r_{ij})$ é atribuída. As relações ainda não consideradas são classificadas como indefinidas, denotadas por $r_{ij} = \emptyset$. A cada nova atribuição de uma variável r_{ij} é necessário verificar a existência de um empacotamento com aquelas relações já atribuídas. A viabilidade de uma solução, parcial ou final, é verificada através da rotina findcoordinates, descrita no algoritmo 3.

Primeiramente o algoritmo findcoordinates inicializa, para todo item $i \in L$, $x_i = 0$, $y_i = 0$ e $z_i = 0$. No laço das linhas 3-36, para cada par de item i, j, verifica-se se existe uma relação atribuída. Se sim, então executa um dos casos das linhas 5-34, deslocando cada caixa para uma posição que respeite a relação considerada. Se a caixa nessa nova posição ultrapassar os limites do contêiner, o algoritmo devolve falso. O laço das linhas 3-36 é executado até que não haja mais alteração no empacotamento. Isto significa que todas as relações atribuídas são respeitadas e nenhuma caixa ultrapassa as dimensões do contêiner. Caso em $|L|$ iterações do laço das linhas 3-36, as posições das caixas sofram mudanças, existe algum ciclo e, portanto, é uma atribuição de relações inválidas. Logo, o algoritmo devolve falso.

Os lemas 4.2.1 e 4.2.2 descritos a seguir mostram a corretude do algoritmo OneBin_General para encontrar um empacotamento viável.

Lema 4.2.1. Se para todo par de itens i, j existe uma relação $r_{ij} \neq \emptyset$ e se uma posição pode ser atribuída pelo algoritmo findcoordinates, então existe um empacotamento.

Demonstração. Nenhum item se sobrepõe a outro, pois se para cada par de itens i e j, alguma relação existe entre eles então, $x_j \geq x_i + w_i$ ou $x_i \geq x_j + w_j$ ou $y_i \geq y_j + h_j$ ou $y_j \geq y_i + h_i$ ou $z_i \geq z_j + d_j$ ou $z_j \geq z_i + d_i$. E nenhum item ultrapassa os limites do contêiner, ou seja se para todo i, temos $x_i + w_i \leq W$, $y_i + h_i \leq H$ e $z_i + d_i \leq D$. Então temos um empacotamento. □

Algoritmo 3: findcoordinates

Entrada: Uma lista de itens L, e uma relação $r_{ij} \in \{$ `l`, `r`, `u`, `a`, `b`, `f`$\}$ ou indefinida, para cada par de itens i, j em L.

Saída: Devolve falso se a atribuição é inviável, e verdade se é viável.

```
1  Para todo i ∈ L, (x_i, y_i, z_i) = (0, 0, 0);
2  para k = 0 até |L| faça
3      para cada i = 1 até |L| faça
4          para cada j = i + 1 até |L| faça
5              caso r_ij = l
6                  se x_j < x_i + w_i então
7                      x_j = x_i + w_i;
8                      se x_j + w_j > W então devolva falso;
9                  fim
10             caso r_ij = r
11                 se x_i < x_j + w_j então
12                     x_i = x_j + w_j;
13                     se x_i + w_i > W então devolva falso;
14                 fim
15             caso r_ij = u
16                 se y_j < y_i + h_i então
17                     y_j = y_i + h_i;
18                     se y_j + h_j > H então devolva falso;
19                 fim
20             caso r_ij = a
21                 se y_i < y_j + h_j então
22                     y_i = y_j + h_j;
23                     se y_i + h_i > H então devolva falso;
24                 fim
25             caso r_ij = f
26                 se z_j < z_i + d_i então
27                     z_j = z_i + d_i;
28                     se z_j + d_j > W então devolva falso;
29                 fim
30             caso r_ij = b
31                 se z_i < z_j + d_j então
32                     z_i = z_j + d_j;
33                     se z_i + d_i > W então devolva falso;
34                 fim
35             fim
36         fim
37     fim
38     se não houve mudanças nas posições então devolva verdade;
39 fim
40 devolva falso; //Existe algum ciclo.
```

Lema 4.2.2. Seja L um conjunto de caixas, qualquer empacotamento $\mathcal{P}'$, que atribui posições (x'_i, y'_i, z'_i) para todo item $i \in L$, possui um empacotamento equivalente que pode ser encontrado através de uma atribuição da relação entre todos os pares de itens.

Demonstração. Para cada par $i,\ j \in L,\ i \neq j$, atribua a primeira relação a seguir que for verdadeira,

- Se $x_j \geq x_i + w_i$, faça $r_{ij} = \texttt{l}$;
- se $x_i \geq x_j + w_j$, faça $r_{ij} = \texttt{r}$;
- se $y_j \geq y_i + h_i$, faça $r_{ij} = \texttt{u}$;
- se $y_i \geq y_j + h_j$, faça $r_{ij} = \texttt{a}$;
- se $z_j \geq z_i + d_i$, faça $r_{ij} = \texttt{b}$;
- se $z_i \geq z_j + d_j$, faça $r_{ij} = \texttt{f}$.

Se nenhuma das condições acima acontecer, i e j se sobrepõem e $\mathcal{P}'$ não pode ser um empacotamento.

Com esse conjunto de relações, o algoritmo findcoordinates encontrará posições (x_i, y_i, z_i) viáveis para cada item no contêiner. □

Se para um dado par de caixas i, j for provado que nenhuma relação pode ser atribuída, o algoritmo realiza um *backtrack*. Caso uma relação seja viável, o algoritmo é chamado recursivamente para o próximo par de caixas. Se todos os pares de caixas $i,\ j$ têm uma relação atribuída, e existe uma atribuição de coordenadas viável tal que nenhuma caixa exceda as dimensões do contêiner, o algoritmo termina com uma solução. Se a busca recursiva não encontrar um empacotamento, OneBin_General devolve falso.

Para tornar o algoritmo mais rápido, redução de domínio e propagação de restrições são aplicados. Redução de domínio é obtido pela abordagem *Forward Check* (FC) proposta por Haralick e Elliot [15]. Para realizar a propagação de restrições é utilizado o algoritmo *Maintaining Arc Consistency* (MAC) proposto por Gaschnig [10]. O algoritmo MAC é um FC mais sofisticado, que a cada passo mantém o problema arco-consistente. O problema é arco-consistente se, para todas as possíveis atribuições de uma relação entre duas caixas i e j, existe uma atribuição viável de coordenadas para as caixas, tal que todas as restrições são satisfeitas. Se isso não é possível, então a relação é removida temporariamente do domínio de $Dom(r_{ij})$. No caso em que um domínio se torna vazio, o problema não pode ser satisfeito e o algoritmo realiza um *backtrack*.

O uso do algoritmo MAC significa que a complexidade de tempo de cada nó na árvore de busca cresce consideravelmente, porém o número de nós investigados decresce correspondentemente. Sabin e Freuder [28] mostraram que o esforço adicional do MAC é compensado quando usado em problemas difíceis.

O algoritmo MAC é utilizado como segue. Para cada chamada recursiva do algoritmo OneBin_General (ou seja, para cada nova atribuição de uma relação), o algoritmo executa sobre todos os pares de caixas i e j com $|Dom(r_{ij})| > 2$ e testa a viabilidade de cada relação $R \in Dom(r_{ij})$. Se o empacotamento se tornar inviável com a relação adicional atribuída, a relação é removida do domínio. Se dessa forma $Dom(r_{ij})$ se torna vazio, podemos concluir que nenhum empacotamento viável pode ser obtido seguindo esse ramo da árvore de busca, então o algoritmo OneBin_General realiza um backtrack. Se sobrar somente uma relação no domínio $Dom(r_{ij})$ então ela é fixada. Tendo uma relação fixada, para manter a arco-consistência pode ser necessário executar uma redução adicional. Contudo, os autores alegam que o custo computacional desse passo não é compensado. Todas as reduções feitas sobre o domínio pela propagação são colocadas em uma pilha, tal que o domínio pode ser facilmente restaurado quando o algoritmo realiza um *backtrack*.

O melhor desempenho do algoritmo OneBin_General, segundo os autores, é alcançado quando ordenando as caixas por volume não crescente. Então, a ramificação é feita sobre os pares de caixas na ordem (1, 2), (1, 3), (2, 3), (1, 4), (2, 4), (3, 4), (1, 5), . . ., ou seja a posição relativa entre a caixa 1 e a caixa 2 que são as maiores caixas é atribuída primeiro. Dessa forma, a posição relativa das caixas de maior volume é estabelecida nos primeiros estágios do algoritmo, e a inviabilidade pode ser detectada mais rapidamente.

Adaptamos esse método para atendar as restrições de ordem, removendo do domínio de cada variável r_{ij} a relação `f`(*front*) se i deve ser descarregado depois de j, e removendo `b`(*behind*) se i deve ser descarregado antes de j.

Lema 4.2.3. Se $r_{ij} \in \{\texttt{l}, \texttt{r}, \texttt{u}, \texttt{a}, \texttt{f}\}$, no empacotamento final, i pode ser removido com um único movimento ortogonal em direção a porta do contêiner. Analogamente, se $r_{ij} \in \{\texttt{l}, \texttt{r}, \texttt{u}, \texttt{a}, \texttt{b}\}$, no empacotamento final, j pode ser removido com um único movimento ortogonal em direção a porta do contêiner.

A demonstração desse lema é análoga a demostração do lema 4.3.1 que veremos na seção seguinte. Denominamos por MPVBK o algoritmo exato e por MPVBK*T*, a heurística obtida limitando o algoritmo MPVBK a uma execução de T segundos.

4.3 Uma formulação através de Programação por Restrições

Nesta seção apresentamos uma formulação em Programação por Restrições para o Problema de Empacotamento Tridimensional utilizando as posições (x, y, z) de cada caixa como variáveis,

diferentemente da formulação por posições relativas entre cada par de caixas apresentada na seção 4.2.

Seja L o conjunto das caixas, denotamos as variáveis do modelo de Programação por Restrições como x_i, y_i e z_i para cada item $i \in L$, que indicam a posição em que será empacotado o item i. Denotamos os domínios das variáveis como todos os pontos possíveis onde um item pode ser posicionado em cada uma das dimensões. Para todo item $i \in I$ definimos

$$Dom(x_i) = [0, \ldots, W - w_i]; \tag{4.1}$$

$$Dom(y_i) = [0, \ldots, H - h_i]; \tag{4.2}$$

$$Dom(z_i) = [0, \ldots, D - d_i]. \tag{4.3}$$

Note que para uma dada caixa i, temos $x_i + w_i \leq W$. Portanto, garantimos que cada caixa não ultrapassa os limites do contêiner. Para assegurar a não sobreposição de itens é necessário que os itens sejam disjuntos em pelo menos um dos eixos X, Y ou Z, Para isso, podemos utilizar a seguinte classe de restrições,

$$\begin{aligned} &\text{para cada par de itens } i, j: \\ &[x_i + w_i \leq x_j] \text{ ou } [x_j + w_j \leq x_i] \text{ ou } [y_i + h_i \leq y_j] \\ &\text{ou } [y_j + h_j \leq y_i] \text{ ou } [z_i + d_i \leq z_j] \text{ ou } [z_j + d_j \leq z_i]. \end{aligned} \tag{4.4}$$

Essa formulação é suficiente para resolver o Problema de Empacotamento Tridimensional no caso onde qualquer ordem de descarregamento é aceitável. A seguir definimos uma forma de modelagem para que seja atendido a restrição de ordem.

4.3.1 Restrição de ordem

Devemos restringir a solução a empacotamentos que obedecem a ordem de descarregamento dos itens. Considere $(L_1, L_2, \ldots, L_k)$ uma sequência de listas de caixas, na ordem em que serão descarregadas. Para $i \in L_u, j \in L_v$, tal que $u < v$, substituímos a restrição 4.4 pela restrição

$$\begin{aligned} [x_i + w_i \leq x_j] \text{ ou } [x_j + w_j \leq x_i] \text{ ou } [y_i + h_i \leq y_j] \\ \text{ou } [y_j + h_j \leq y_i] \text{ ou } [z_j + d_j \leq z_i]. \end{aligned} \tag{4.5}$$

Lema 4.3.1. Dado duas caixas i e j, se [$x_i + w_i \leq x_j$ ou $x_j + w_j \leq x_i$ ou $y_i + h_i \leq y_j$ ou $y_j + h_j \leq y_i$ ou $z_j + d_j \leq z_i$], então i e j não se sobrepõem, e i pode ser retirada com um único movimento ortogonal em direção a frente do contêiner sem ser bloqueado pelo item j.

Demonstração. Conforme descrito na seção 2.1.3, a caixa i pode ser retirada com um movimento ortogonal em direção a frente do contêiner sem ser bloqueado pelo item j, se $\mathcal{R}^z(i) \cap \mathcal{R}(j) = \emptyset$. Para mostrar que isso acontece vamos considerar o termo $(x_i + w_i \leq x_j)$ da restrição. Para os demais termos, a prova é análoga.

Se $x_i + w_i \leq x_j$, então a região ocupada por i e pelo espaço utilizado para realizar o movimento em direção a frente do contêiner é dada por $\mathcal{R}^z(i) = [x_i, x_i + w_i \leq x_j) \times [y_i, y_i + h_i) \times [z_i, D)$, e a região $\mathcal{R}(j)$ ocupada por j é definido como $\mathcal{R}(j) = [x_j, x_j + w_j) \times [y_j, y_j + h_j) \times [z_j, z_j + d_j)$ (vide seção 2.1.3). Como $[x_i, x_i + w_i \leq x_j) \cap [x_j, x_j + w_j) = \emptyset$ as regiões são disjuntas, portanto $\mathcal{R}^z(i) \cap \mathcal{R}(j) = \emptyset$.

Dessa forma, se qualquer um dos termos da restrição for verdadeiro, i e j não se sobrepõem, e i pode ser retirado com um único movimento ortogonal em direção a frente do contêiner sem ser bloqueado pelo item j. □

As caixas pertencentes a uma mesma lista continuam com a restrição (4.4), pois não existe restrição de ordem entre itens da mesma lista, sendo necessário somente garantir a não sobreposição.

4.3.2 Redução de Domínio

A formulação acima pode ser melhorada utilizando uma técnica de redução de domínio, que visa diminuir o número de possibilidades onde um item pode ser posicionado. Para isso, em cada uma das dimensões, usaremos os *pontos de discretização* descrito por Herz [16].

Definição 4.3.1. Seja L um conjunto de caixas. Um ponto de discretização da largura de L é um valor $x' < W$ que pode ser obtido por qualquer combinação de $w_1, w_2, \ldots, w_n$. Denotamos por $\mathcal{X}$ o conjunto de pontos de discretização da largura. Formalmente,

$$\mathcal{X} = \left\{ x' : x' = \sum_{i \in S} w_i, \forall S \subseteq L \right\}$$

Analogamente, denotamos pontos de discretização para a altura e para a profundidade, sendo, respectivamente, $\mathcal{Y}$ e $\mathcal{Z}$ os conjuntos desses pontos. O algoritmo XDP encontra os pontos de discretização da largura. Inicialmente, na linha 3 a coordenada 0 é adicionada à $\mathcal{X}$. Depois, no laço das linhas 4-10, iteramos para cada item de I, iniciando um conjunto auxiliar $\mathcal{X}'$, e para cada coordenada $\bar{x} \in \mathcal{X}$ verificamos se o ponto $\{\bar{x} + w_i\}$ não excede o limite do contêiner, adicionando-o em $\mathcal{X}'$. No final do laço, adicionamos os pontos de $\mathcal{X}'$ em $\mathcal{X}$. Por fim, o algoritmo devolve $\mathcal{X}$.

Algoritmo 4: XDP

Entrada: um conjunto dos itens I;
saída: um conjunto $\mathcal{X}$ de coordenadas x;
$\mathcal{X} \leftarrow \{0\}$
para cada $i \in I$ **faça**
    $\mathcal{X}' \leftarrow \{\}$
    para cada $\bar{x} \in \mathcal{X}$ **faça**
        se $\bar{x} + w_i < W$ **então** $\mathcal{X}' \leftarrow \mathcal{X}' \cup \{\bar{x} + w_i\}$;
    fim
    $\mathcal{X} \leftarrow \mathcal{X} \cup \mathcal{X}'$
fim
devolva $\mathcal{X}$;

Dessa forma, atribuímos para as variáveis x_i, y_i e z_i de cada item $i \in I$, o seguinte domínio.

$$Dom(x_i) = \{x' \in \mathcal{X} | x' + w_i < W\} \tag{4.6}$$
$$Dom(y_i) = \{y' \in \mathcal{Y} | y' + h_i < H\} \tag{4.7}$$
$$Dom(z_i) = \{z' \in \mathcal{Z} | z' + d_i < D\} \tag{4.8}$$

Lema 4.3.2. Qualquer empacotamento possui um empacotamento equivalente onde cada item está posicionado em um ponto de discretização.

Demonstração. Uma prova semelhante foi apresentada por Christofides e Whitlock [3] e por Herz [16] para o caso bidimensional. É fácil ver que qualquer empacotamento possui um equivalente em que nenhum item pode ser movido para trás, para esquerda ou para baixo. Nesse empacotamento, cada item tem associado um conjunto de itens que impedem que o item seja deslocado em cada uma das direções. Dessa forma, para cada item i empacotado em um ponto (x_i, y_i, z_i), há um conjunto de itens $\{l_1, \ldots, l_k\}$, tal que $\sum_{j=1,\ldots,k} w_{l_k} = x_i$, que previne que o item i seja deslocado para esquerda. Analogamente, há um conjunto de itens que previne i de se deslocar para trás e para baixo. □

Denominamos por CP o algoritmo exato utilizando essa formulação e por CP*T*, a heurística obtida limitando o algoritmo CP a uma execução de T segundos.

4.4 Heurísticas para o 3DPLC

Nesta seção apresentamos duas heurísticas para o Problema de Empacotamento Tridimensional com Restrição de Ordem. A primeira é baseada na heurística apresentada por George e

Robinson [19], que utiliza uma abordagem de construção em camadas (*Wall-Building*). A segunda é uma heurística de níveis baseada no algoritmo híbrido HFF (*Hibrid First Fit*) para o empacotamento bidimensional.

4.4.1 Heurística de George e Robinson

A heurística, denominada por GR, utiliza uma pilha S de espaços disponíveis dentro do contêiner, ainda não utilizados no empacotamento. Inicialmente, S recebe um único espaço, correspondendo a totalidade do contêiner. Para garantir que a restrição de ordem do empacotamento seja satisfeita, o algoritmo atribui a posição de um item somente se os itens que serão descarregados posteriormente já estão com suas posições definidas. Ou seja, dado uma sequência $\mathcal{L}$ de listas de caixas $(L_1, \ldots, L_k)$, o empacotamento da lista L_u será realizado somente depois que todas as listas L_v, com $v < u$, forem empacotadas.

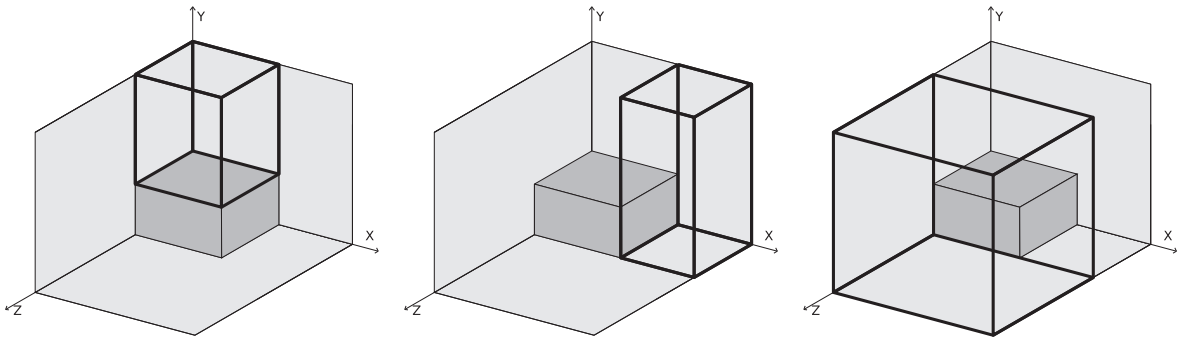
Definimos um espaço s como uma região $[x'_s, x''_s) \times [y'_s, y''_s) \times [z'_s, z''_s)$ contida no contêiner. Uma caixa i pode ser empacotada em um espaço s, se $w_i \leq x''_s - x'_s$, $h_i \leq y''_s - y'_s$ e $d_i \leq z''_s - z'_s$. Esse empacotamento é sempre feito no canto inferior, esquerdo e traseiro do espaço, ou seja, se (x_i, y_i, z_i) será a posição da caixa i no empacotamento, o algoritmo atribui $x_i = x'_s$, $y_i = y'_s$ e $z_i = z'_s$.

O algoritmo também utiliza um conjunto de espaços rejeitados R, que contém espaços que não foram utilizados. Os espaços em R permanecem a disposição para serem fundidos com outros espaços. Durante a execução do algoritmo um espaço s é fundido a um espaço rejeitado $r \in R$ se $[x'_r, x''_r) \times [y'_r, y''_r) \subseteq [x'_s, x''_s) \times [y'_s, y''_s)$, e se $z''_r = z'_s$. Dessa forma s é expandido e z'_s assume valor z'_r. Sobras do espaço r são descartadas.

O algoritmo 5 apresenta a heurística GR. A função $push(S, s)$ coloca o espaço s no topo da pilha S, e a função $pop(S)$ extrai o espaço no topo da pilha. O algoritmo recebe uma sequência de listas de caixas $\mathcal{L}$ e um contêiner de dimensões (W, H, D), devolvendo um empacotamento $\mathcal{P}$ que atribui posições (x_i, y_i, z_i) para cada caixa i, ou devolve falso caso esse empacotamento não seja encontrado.

No início do algoritmo, ordenamos cada lista de $\mathcal{L}$, pelo valor da menor dimensão de cada caixa, ou seja, pelo valor $\min\{w_i, h_i, d_i\}$, de forma não crescente. Na linha 2 colocamos na pilha o espaço do contêiner, e na linha 3 inicializamos a variável j que indica a lista de caixas atual que deve ser empacotada. O laço externo itera até que todas as caixas tenham sido empacotadas, no início desse laço, na linha 5, tiramos do topo da pilha S um espaço s, e procuramos espaços rejeitados em R que possam ser fundidos a s, conforme descrito anteriormente.

O laço interno das linhas 7-12 executa, caso nenhuma caixa de L_j couber em s, adicionando s aos espaços rejeitados, retirando um novo espaço da pilha e fundindo-o com espaços de R. Escolhido um espaço s, removemos a primeira caixa i de L_j que cabe em s, atribuindo a posição de i no empacotamento $\mathcal{P}$.



(a) Superior (b) Lateral (c) Frontal

Figura 4.4: Novos espaços gerados.

Algoritmo 5: GR

Entrada: Uma sequência de listas de caixas $\mathcal{L} = \{L_1, \ldots, L_k\}$, e um contêiner de dimensões (W, H, D).

Saída: Devolve um empacotamento $\mathcal{P}$ que atribui posições (x_i, y_i, z_i) para cada caixa i. Devolve falso se não encontrar um empacotamento.

Ordenar cada lista de $\mathcal{L}$, pela menor dimensão de cada caixa, de forma não crescente;
$push(S, [0, W) \times [0, H) \times [0, D))$;
$j = 1$;
enquanto $j \leq k$ **faça**
    $s = pop(S)$;
    Fundir se possível s com espaços rejeitados de R;
    enquanto *nenhuma caixa de* L_j *couber em* s **faça**
        Adiciona s nos espaços rejeitados;
        se $S \neq \emptyset$ **então devolve** falso;
        $s = pop(S)$;
        Fundir se possível s com espaços rejeitados de R;
    fim
    Selecione e remove uma caixa i de L_j que cabe em s;
    $x_i = x'_s$; $y_i = y'_s$; $z_i = z'_s$;
    $push(S, [x'_s, x''_s) \times [y'_s, y''_s) \times [z'_s + d_i, z''_s))$;
    $push(S, [x'_s + w_i, x''_s) \times [y'_s, y''_s) \times [z'_s, z'_s + d_i))$;
    $push(S, [x'_s, x'_s + w_i) \times [y'_s + h_i, y''_s) \times [z'_s, z'_s + d_i))$;
    se $L_j = \emptyset$ **então** $j = j + 1$;
fim
devolve $\mathcal{P}$.

Quando uma caixa i é empacotada em um espaço s, três espaços menores são gerados. O espaço superior à caixa até a parte superior do espaço, $[x'_s, x'_s + w_i) \times [y'_s + h_i, y''_s) \times [z'_s, z'_s + d_i)$. Ao lado direito desse espaço e da caixa, é gerado o espaço lateral, $[x'_s + w_i, x''_s) \times [y'_s, y''_s) \times [z'_s, z'_s + d_i)$, e o espaço afrente desses espaços e da caixa, $[x'_s, x''_s) \times [y'_s, y''_s) \times [z'_s + d_i, z''_s)$. Veja figura 4.4.

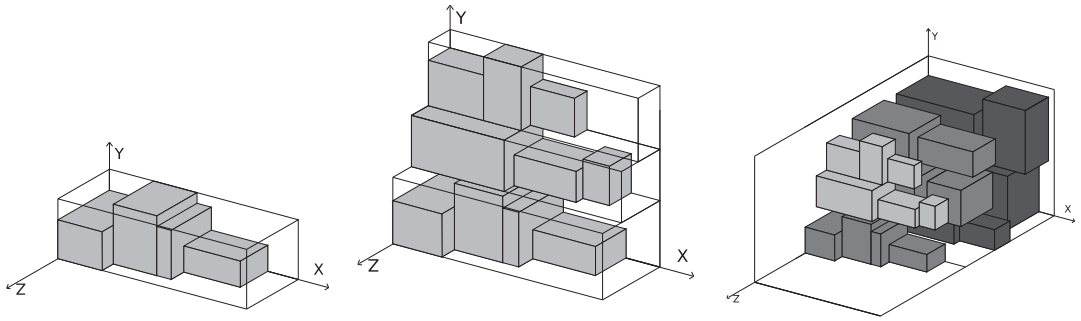
Nas linhas 15, 16 e 17 o algoritmo coloca na pilha os espaços frontal, lateral e superior. Por fim na linha 18 o algoritmo verifica se todas as caixas da lista atual foram empacotadas. Se sim, parte para a próxima lista incrementando j.

4.4.2 Heurística de Empacotamento em Níveis

A heurística, denominada por 3HFF, recebe uma sequência de listas de caixas $\mathcal{L} = \{L_1, \ldots, L_k\}$ e um contêiner de dimensões (W, H, D). Então constrói um empacotamento em níveis. Inicialmente, as caixas de cada cliente são empacotadas uma ao lado da outra em prateleiras, de largura máxima W, conforme figura 4.5a. Denominamos por S_j uma sequência de prateleiras formados por itens de L_j. Não permitimos que itens de diferentes listas compartilhem a mesma prateleira. O empacotamento das caixas de L_j é feito da seguinte forma: inicialmente, as caixas de L_j são ordenadas pela profundidade de forma não crescente, então cada caixa $i \in L_j$ é alocada na primeira prateleira de S_j, em que a soma das larguras dos itens não ultrapasse W. Quanto uma caixa não cabe em nenhuma das prateleiras existentes, uma nova prateleira é gerada em S_j. Essa estratégia é conhecida como *first fit*. A profundidade e a altura da prateleira é determinada pelas caixas de maior profundidade e maior altura, respectivamente.

As prateleiras geradas são, então, empacotadas em camadas (veja figura 4.5b), usando a mesma estratégia de *first fit*. Geramos uma sequência C de camadas, de altura máxima H, e para cada $j = 1, \ldots, k$, empacotamos as prateleiras de S_j na primeira camada de C cuja soma das alturas das prateleiras não ultrapasse H. Dentro do contêiner essas camadas são posicionadas na ordem em que são geradas. Permitimos que as prateleiras do cliente i sejam empacotadas somente a partir da última camada de prateleiras do cliente $i - 1$. Sempre que uma prateleira não couber em qualquer camada viável, geramos uma nova camada.

O algoritmo 6 descreve a heurística 3HFF. Denotamos por h'_s, e h'_c, as alturas de uma prateleira s e de uma camada c , e por d'_c a profundidade da camada c. No laço das linhas 1-4, o algoritmo ordena os itens de cada lista de $\mathcal{L}$ e gera as sequências de prateleiras, conforme descrito anteriormente. Na linha 5, o algoritmo gera a primeira camada, ainda sem nenhuma prateleira, e na linha 6 o algoritmo inicializa a variável auxiliar $last$, que indica o índice da última camada gerada.



(a) Prateleira (b) Camada (c) Contêiner

Figura 4.5: Prateleira, Camada e Contêiner.

Algoritmo 6: 3HFF

Entrada: Uma sequência de listas de caixas $\mathcal{L} = \{L_1, \ldots, L_k\}$, e um contêiner de dimensões (W, H, D).

Saída: Devolve um empacotamento ou falso, caso não encontre algum.

1 **para cada** $i = 1, \ldots, k$ **faça**
2 ordena L_i por profundidade não crescente;
3 empacota L_i em prateleiras usando a estratégia *first fit* gerando S_i, a sequência de prateleiras obtidas;
4 **fim**
5 Gera primeira camada c_0 vazia em C;
6 $last = 0$;
7 **para cada** $i = 1, \ldots, k$ **faça**
8 $feas = last$;
9 **para cada** *prateleira* s *em* S_i **faça**
10 **para** $j = feas, \ldots, last$ **faça**
11 **se** $h'_s + h'_{c_j} < H$ **então**
12 aloca prateleira s na camada c_j;
13 **vá para** linha; 19
14 **fim**
15 **fim**
16 //Não coube em nenhuma camada;
17 $last = last + 1$;
18 gera nova camada c_{last}, alocando s;
19 **fim**
20 **fim**
21 **se** $\sum_{c \in C} d'_c > D$ **então devolve** falso;
22 **devolve** $\mathcal{P}$.

No laço das linhas 7-20 a variável i indica o índice da sequência de prateleiras sendo considerado atualmente. A variável auxiliar $feas$ indica a primeira camada na qual as prateleiras de S_i podem ser alocadas. Para cada prateleira $s \in S_i$, no laço das linhas 9-14, o algoritmo procura entre as camadas viáveis a primeira em que s pode ser alocada. Se s não pode ser alocada em alguma das camadas viáveis, uma nova camada é gerada com s.

Por fim, se a soma das profundidades das camadas ultrapassa o limite do contêiner, o algoritmo devolve falso, senão devolve o empacotamento encontrado.

Capítulo 5

Um Algoritmo Exato para o 3DTSPL

No Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional (3DTSPL) precisamos encontrar uma rota que passe por todos os clientes, e que seja possível alocar as caixas de cada cliente no contêiner do veículo. Esse empacotamento deve ser feito de forma que a cada visita, as caixas a serem entregues possam ser descarregadas sem serem obstruídas pelas caixas remanescentes.

Neste capítulo propomos um algoritmo exato para a resolução do 3DTSPL. O algoritmo consiste em um método de *branch-and-cut* que insere desigualdades violadas no problema. Na seção 5.1 reforçamos a formulação em programação linear inteira do problema que vamos considerar. Nas seções 5.2 e 5.3 apresentamos como são encontradas cada uma das desigualdades violadas. Por fim, na seção 5.4 apresentamos como compôr esses procedimentos em uma rotina de separação.

5.1 Formulação para o 3DTSPL

Conforme apresentado na seção 2.2, considere $\mathcal{I}_{\text{3DTSPL}}$ o conjunto de instâncias do problema 3DTSPL definido pelo conjunto de tuplas $(V, E, c, B, \mathbb{L})$, em que $G = (V, E, c)$ é o grafo de entrada, B um contêiner de dimensões (W, H, D), e $\mathbb{L}$ um conjunto $\{L_1, \ldots, L_k\}$, onde cada L_v é uma lista de caixas para cada vértice $v \in V \backslash \{0\}$.

Dado uma instância $I \in \mathcal{I}_{\text{3DTSPL}}$, denotamos por $\mathcal{R}_I$ o conjunto de todas as possíveis rotas dessa instância que tem algum empacotamento de $\mathbb{L}$ em B que respeita a ordem em que serão visitados os clientes nessas rotas. Seja $\delta(S)$ o conjunto de arestas de um corte S, conforme descrito na seção 2.1.1, e por x_e a variável que recebe 1 se a aresta e pertence a solução e 0, caso contrário. A formulação do problema 3DTSPL é apresentada a seguir.

$$\text{Minimize} \sum_{e \in E} c_e x_e,$$

sujeito a

$$\sum_{e \in \delta(v)} x_e = 2 \qquad \forall v \in V\,, \tag{5.1}$$

$$\sum_{e \in \delta(S)} x_e \geq 2 \qquad \forall S \subset V, \quad S \neq \emptyset\,, \tag{5.2}$$

$$\sum_{e \in R} x_e \leq n-1 \qquad \forall R \notin \mathcal{R}_I\,, \tag{5.3}$$

$$x_e \in \{0,1\} \qquad \forall e \in E\,. \tag{5.4}$$

A classe de restrições (5.4) limita as variáveis x_e , para cada $e \in E$, aos valores 0 ou 1. A classe de restrições (5.1) garante que em cada vértice incidem exatamente duas arestas. A classe de restrições (5.2) garante que a solução será um circuito hamiltoniano do grafo de entrada. Nos referimos à esta classe de restrições por *restrições de conectividade*. As restrições da classe (5.3) são as *restrições de empacotamento* e restringem o conjunto de rotas apenas para aquelas que têm um empacotamento que respeita a ordem respectiva rota.

O número de restrições de conectividade (5.2) é determinado pela quantidade de subconjuntos possíveis de se formar com os elementos de V. A quantidade desses subconjuntos é exponencial, e para verificar isso considere $\mathscr{S}$ o conjunto de todos os subconjuntos próprios de V, $S \subset V$ um dos elementos de $\mathscr{S}$, e considere um vértice v em V. O vértice v pode ou não estar em S, e da mesma forma, cada um dos vértices de V. Assim o número total de combinações dos elementos de V é $2^{|V|}$, portanto $|\mathscr{S}| = 2^{|V|}$ que é exponencial.

O número de restrições de empacotamento (5.3) também pode ser exponencial. Para verificar isso, suponha um grafo G completo, ou seja, para cada par de vértices u e v em V, a aresta $\{u, v\}$ pertence à E. Denotamos por k o número de clientes do problema, isto é, $k = |V| - 1$. Como G é completo, existem k possíveis destinos para o veículo visitar partindo do depósito, $k-1$ possíveis destinos para o veículo após visitar o primeiro cliente e assim por diante. Então, existem $k!$ possíveis rotas para o veículo. Dessa forma, podemos dizer que para qualquer grafo G exitem $O(k!)$ rotas distintas. Além disso, para cada rota, verificar se existe ou não um empacotamento de $\mathbb{L}$ em B que respeite a ordem dessa rota é um problema NP-Difícil.

Seria inviável considerar a priori todas as restrições das classes (5.2) e (5.3). Por isso, considere a seguinte relaxação da formulação apresentada, onde removemos as restrições (5.2) e (5.3), e relaxamos a restrição de integralidade.

$$\text{Minimize} \sum_{e \in E} c_e x_e,$$

sujeito a

$$\sum_{e \in \delta(v)} x_e = 2 \qquad \forall v \in V\,, \tag{5.5}$$

$$x_e \in [0,1] \qquad \forall e \in E\,. \tag{5.6}$$

Utilizando essa relaxação podemos resolver o problema utilizando o método *branch-and-cut*, apresentado no capítulo 3. Para isso, alimentamos o algoritmo de *branch-and-bound* com rotinas de separação, inserindo restrições violadas do tipo (5.2) e (5.3) sempre que é detectado uma solução candidata que viola alguma dessas restrições. Os cortes que separam restrições fracionárias de soluções inteiras são inseridas pelo próprio resolvedor. Nas seções seguintes apresentaremos uma forma de detectar quando uma solução para a relaxação viola as desigualdades (5.2) e (5.3).

5.2 Restrições de Conectividade

Como rotina de separação das restrições (5.2) utilizamos cortes de conexidade, obtidas a partir de cortes mínimos, definidos na subseção 5.2.1. Inicialmente, executamos o procedimento da árvore de Gomory-Hu [13, 14], apresentado na subseção 5.2.2, que fornece todos os cortes mínimos entre vértices através de $|V|-1$ chamadas para o algoritmo de corte mínimo (ao invés de se chamar para cada um dos $\binom{|V|}{2}$ pares possíveis de vértices). Com isso, inserimos todos os cortes violados da árvore. Este processo é combinado com buscas de corte mínimo simples e repetido até que todas as restrições em (5.2) sejam satisfeitas.

5.2.1 Corte Mínimo

Considere um grafo em que cada aresta tem uma capacidade associada. É comum entendermos esse tipo de grafo como uma rede hidráulica, sendo as arestas os canos e a capacidade corresponde a quantidade de água que cada cano suporta. Considere, então, $G = (V, E)$ um grafo desse tipo e a função $w : E \to \mathbb{Q}^+$ que atribui uma capacidade a cada aresta de E.

Definição 5.2.1. Definimos um *s-t corte* como um subconjunto S dos vértices em V, tal que $s \in S$, $t \notin S$ e $\delta(S)$ é o conjunto das arestas que possuem somente uma extremidade em S. Também definimos $C(S)$ como a capacidade do corte S, que é dado pela soma das capacidades das arestas de $\delta(S)$, ou seja,

$$C(S) = \sum_{e \in \delta(S)} w_e.$$

Definição 5.2.2. Um *s-t corte mínimo* é um s-t corte S tal que a soma das capacidades das arestas em $\delta(S)$ seja mínimo.

A figura 5.1 exemplifica um corte entre E e A. O conjunto $S = \{E, C\}$ é o corte mínimo separando os vértices E e A, sendo $\delta(S)$ o conjunto das arestas destacadas.

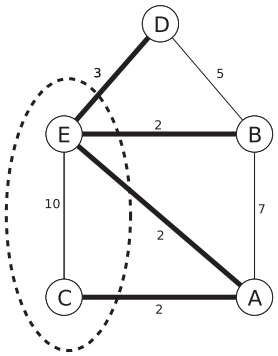


Figura 5.1: Exemplo de corte entre E e A.

Existem na literatura diversos algoritmos para encontrar cortes mínimos. Nesse trabalho utilizamos um dos métodos mais eficientes, proposto por Goldberg e Tarjan [12].

5.2.2 Árvore de Gomory-Hu

O método de Gomory-Hu foi apresentado por Gomory e Hu [13]. Mais tarde foi simplificado por Gusfield [14]. Esse método encontra o valor de cortes mínimos entre todos os pares de vértices de um grafo $G = (V, E)$, fazendo $|V| - 1$ chamadas a um algoritmo que devolve um s-t corte mínimo em um grafo. O método constrói uma árvore denominada Árvore de Gomory-Hu que pode ser utilizada para encontrar desigualdades de conectividade violadas. Para formalizar a definição de Árvore de Gomory-Hu, precisamos da definição de caminho, dada a seguir.

Definição 5.2.3. Dado um grafo $G = (V, E)$ e dois vértices $s, t \in V$. Um s-t caminho P em G é um conjunto de arestas $\{\{v_0, v_1\}, \ldots, \{v_{r-1}, v_r\}\}$, tal que $v_i \neq v_j$, para todo $i \neq j$, $v_0 = s$, $v_r = t$, e $\{v_i, v_{i+1}\} \in E$, $i = 0 \ldots r - 1$.

Definição 5.2.4. Seja $G = (V, E, w)$ um grafo, em que w_e é a capacidade de cada aresta $e \in E$. Seja $G^T = (V^T, E^T, w^T)$ uma árvore com $V^T = V$, S_{st} um s-t corte mínimo em G e P_{st} as arestas do s-t caminho em G^T para s, $t \in V$. Dizemos que G^T é uma Árvore de Gomory-Hu de G se $C(S_{st}) = \min\{w_e^T : e \in P_{st}\}$ para todo $s, t \in V$.

Algoritmo 7: Árvore de Gomory-Hu

Entrada: Um grafo $G = (V, E, w)$ com $V = \{0, \ldots, n\}$.
Saída: Uma Árvore de Gomory-Hu $G^T = (V^T, E^T, w^T)$.

```
1  p[v] = 0, para todo v ∈ V;
2  para s = 1 até |V| faça
3      calcule um corte mínimo S entre os vértices s e t = p[s];
4      e faça C(S) a capacidade do corte;
5      fl[s] = C(S);
6      para i = 0 até |V| faça
7          se i ≠ s e i ∈ S e p[i] = t então p[i] = s;
8      fim
9      se p[H] ∈ S então
10         p[s] = p[H];
11         p[H] = s;
12         fl[s] = fl[H];
13         fl[H] = C(S);
14     fim
15 fim
16 V^T = V;
17 para i = 1 até |V| faça
18     E^T = E^T ∪ {{i, p[i]}};
19     w^T_{i,p[i]} = fl[i];
20 fim
21 devolva G^T.
```

Descrevemos o método de Gomory-Hu, na versão apresentada por Gusfield [14]. A ideia do método é a seguinte. Inicialmente unimos todos os vértices de G em um único supervértice $\mathcal{S}$. Esse supervértice é dividido originando dois novos supervértices, que são novamente divididos. Essa divisão é feita até que cada supervértice contenha apenas um vértice. Cada supervértice possui um representante $r(\mathcal{S})$ que é um dos vértices de $\mathcal{S}$. Denotamos um supervértice contendo v por $\mathcal{S}_v$.

Quando dividimos um supervértice $\mathcal{S}$, computamos um corte mínimo qualquer em G entre $r(\mathcal{S})$ e um outro vértice qualquer $v \in \mathcal{S}$. Os vértices de $\mathcal{S}$ que ficarem do lado de v nesse corte, formam um novo supervértice $\mathcal{S}_v$ com representante v. Os outros nós em $\mathcal{S}$ permanecem em $\mathcal{S}_{r(\mathcal{S})}$ com representante $r(\mathcal{S})$. Se um supervértice $\mathcal{S}'$ é vizinho de $\mathcal{S}$ em G^T antes da divisão, e $r(\mathcal{S}')$ está do lado de v no corte, então a aresta $\{\mathcal{S}, \mathcal{S}'\}$ é substituída pela aresta $\{\mathcal{S}_v, \mathcal{S}'\}$.

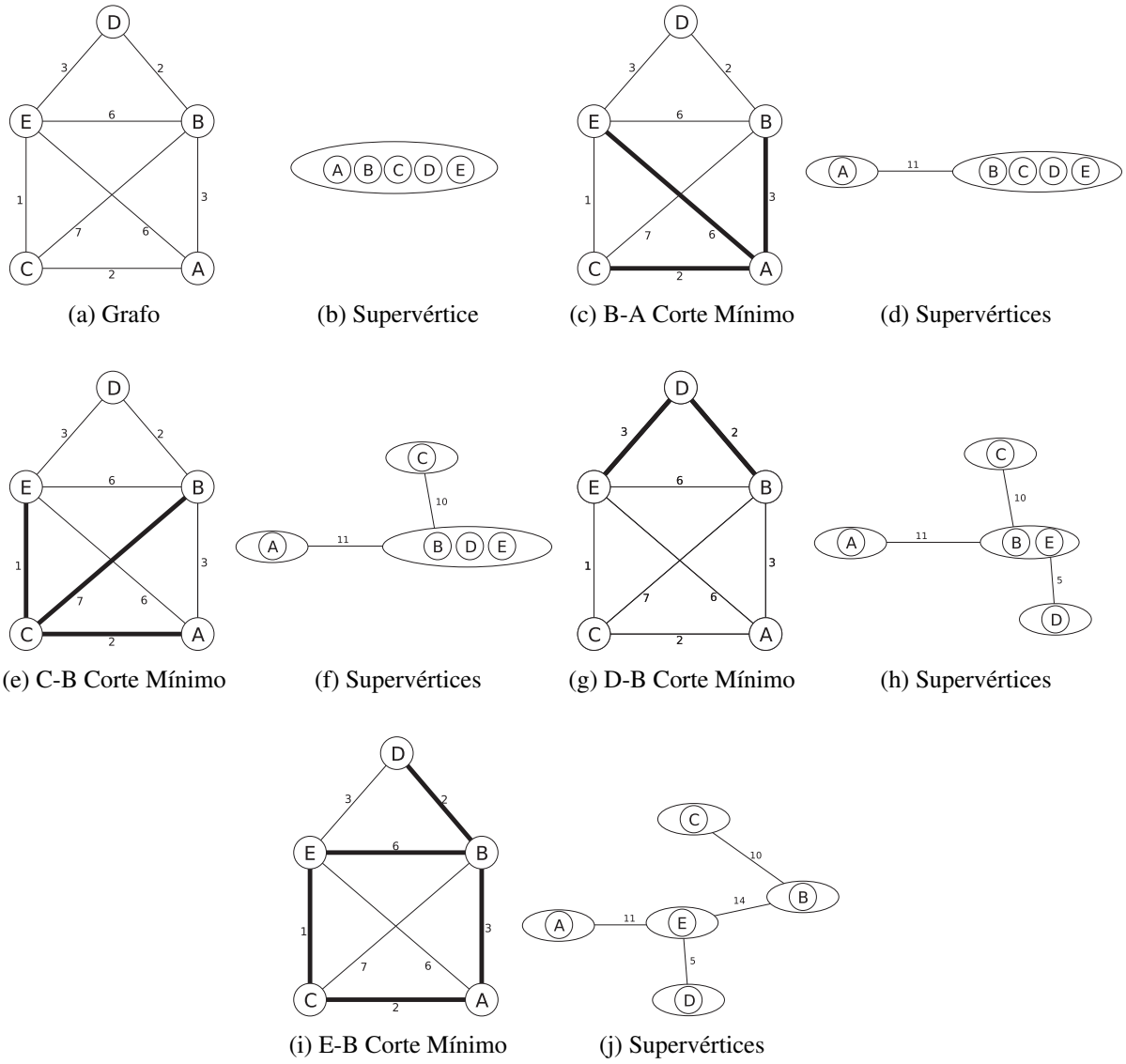
O algoritmo 7 foi adaptado de Gusfield [14], e encontra uma árvore de Gomory-Hu. No início de cada iteração $i = 0, \ldots, |V|$, denotamos por $p[i]$ o representante do supervértice que

contém o vértice i. Na linha 1 inicializamos $p[v]$ colocando todos os vértices no mesmo supervértice representado pelo vértice 0. No laço das linhas 2-15 a divisão do supervértice será realizada sempre escolhendo o próximo vértice na ordem numérica.

Na linha 3 do algoritmo, encontramos o corte mínimo S entre o vértice selecionado s e o vértice t representante do supervértice $\mathcal{S}$ que contêm s. O valor do corte é $C(S)$ que será guardado na variável $fl[s]$. No laço das linhas 6-8 é feita a divisão do supervértice. Os vértices em S do supervértice $\mathcal{S}$ estarão no novo supervértice $\mathcal{S}_s$. Nas linhas 9-14 é feito o ajuste da aresta quando $p[H]$ pertence ao corte S. Ao final do algoritmo, as arestas de G^T são os pares $\{i, p[i]\}$ com capacidade $fl[i]$ para $i = 1, \ldots, |V|$.

A prova de corretude desse algoritmo é bastante extensa e pode ser verificada com detalhes em [14].

A figura 5.2 exemplifica a aplicação do algoritmo no grafo da figura 5.2a. Inicialmente é gerado o supervértice $\{A, B, C, D, E\}$ como mostra a figura 5.2b, com vértice representante A. Depois é escolhido o vértice B, e um corte mínimo é encontrado separando os vértices A e B como mostra a figura 5.2c. Então, é gerado o supervértice $\{B, C, D, E\}$ com representante B e uma aresta com capacidade igual ao valor do corte mínimo, conforme a figura 5.2d. Posteriormente encontramos o corte mínimo da figura 5.2e separando os vértices C e B. O supervértice $\{C\}$ é gerado conectado com o supervértice de B por uma aresta de capacidade 10, conforme a figura 5.2f. Em seguida, o vértice D é selecionado e encontramos o corte mínimo mostrado na figura 5.2g que separa D de B. O supervértice $\{D\}$ é gerado, conforme a figura 5.2h. Por fim, procuramos o corte mínimo que separa E e B, que é apresentado na figura 5.2i, e o último supervértice é gerado, encerrando o algoritmo. A árvore de Gomory-Hu é representada na figura 5.2j.



(a) Grafo (b) Supervértice (c) B-A Corte Mínimo (d) Supervértices

(e) C-B Corte Mínimo (f) Supervértices (g) D-B Corte Mínimo (h) Supervértices

(i) E-B Corte Mínimo (j) Supervértices

Figura 5.2: Um exemplo de construção de Árvore de Gomory-Hu.

5.3 Restrições de Empacotamento

Sempre que uma rota for encontrado na árvore, executamos as rotinas de separação das restrições (5.3). Para isso, utilizaremos os algoritmos de empacotamento apresentados no capítulo 4. Aplicamos o algoritmo MPV e MPVBK com limite de tempo. Caso não se consiga obter um empacotamento que satisfaça a ordem da rota, executamos o algoritmo exato CP.

Além disso, é possível obter um empacotamento por robô exato, utilizando somente o al-

goritmo MPV. Também é possível obter apenas uma solução heurística, em que o tempo computacional pode ser reduzido, mas o custo da rota pode não ser ótimo. Testamos diferentes combinações desses algoritmos e reportamos os resultados no capítulo 6.

5.4 Rotina de separação para o 3DTSPL

Em cada iteração da otimização em um nó da árvore de *branch-and-bound*verificamos se existem restrições da classe (5.2) violadas, ou seja, se existe $S \subset V$ tal que $\sum_{e \in \delta(S)} x_e < 2$. Para descobrir essas violações, primeiramente encontramos uma Árvore de Gomory-Hu $G^T = (V^T, E^T, w^T)$. Então, para cada aresta $e = \{s, t\} \in E^T$, em que $w_e^T < 2$, encontramos um s-t corte mínimo S e adicionamos a desigualdade violada $\sum_{e \in \delta(S)} x_e \geq 2$. Note que esse procedimento pode ser realizado mesmo quando os valores de x_e são fracionários.

Se nenhuma restrição de corte estiver violada, mas ainda existirem valores de x_e fracionários, continuamos com o *branch-and-cut*, adicionando os cortes providos pelo resolvedor. Quando atingimos uma solução inteira, ou seja, $x_e \in \{0, 1\}$ para toda aresta $e \in E$, determinamos uma rota R. Então procuramos um empacotamento que respeite a ordem em que são visitados os clientes em R. Se tal empacotamento não existir, ou seja, se $R \notin \mathcal{R}_I$, adicionamos a restrição da classe (5.3) correspondente, $\sum_{e \in R} x_e \leq |V| - 1$.

O algoritmo 8 apresenta como são encontrados as desigualdades violadas. O algoritmo recebe a solução atual da relaxação, o grafo de entrada, um algoritmo $\mathcal{A}$ para o 3DPLC, o conjunto de listas de caixas e o contêiner B. O algoritmo devolve o conjunto $\mathcal{D}$ de desigualdades encontradas. Na linha 2 é gerado o grafo auxiliar $G' = (V', E', w')$, tal que $V' = V$ e $E' = E$. São atribuídas as capacidades nas arestas com o valor da solução (da relaxação) recebido, ou seja, $w'_e = x_e$ para cada $e \in E$.

A seguir, na linha 3, o algoritmo obtém uma árvore de Gomory-Hu, utilizando o procedimento descrito na subseção 5.2.2. Nas linhas 5-7 o algoritmo adiciona a desigualdade $\sum_{e \in \delta(S)} x_e \geq 2$, para cada aresta da árvore de Gomory-Hu cuja capacidade seja menor que 2. Se nenhuma aresta for violada e a solução for inteira, nas linhas 12-14, utilizando o algoritmo $\mathcal{A}$, é feita a busca por um empacotamento que respeite a rota $\mathcal{R}$ formada. Se tal empacotamento não existir, o algoritmo adiciona a desigualdade $\sum_{e \in \mathcal{R}} x_e \leq |V| - 1$, que vai eliminar essa rota das soluções viáveis do problema. Consideramos que $\mathcal{A}$ devolve $\emptyset$ caso não encontre nenhum empacotamento.

Caso nenhuma das arestas da árvore de Gomory-Hu tenha capacidade menor que 2, mas a solução ainda seja fracionária, nenhuma desigualdade é devolvida e o algoritmo de *branch-and-cut* continua com a adição de cortes encontrados pelo resolvedor para eliminar soluções fracionárias.

Algoritmo 8: Separação

Entrada: Um grafo $G = (V, E)$. os valores x obtidos na resolução da relaxação, um algoritmo $\mathcal{A}$ para o 3DPLC, um contêiner B e um conjunto $\mathbb{L} = \{L_1, \ldots, L_k\}$ de listas de caixas.

Saída: Um conjunto de desigualdades violadas $\mathcal{D}$.

$\mathcal{D} \leftarrow \emptyset$;
seja $G' = (V', E', w')$, com $V' = V$, $E' = E$, e $w'_e = x_e$ para cada $e \in E$;
seja $G^T = (V^T, E^T, w^T)$ uma árvore de Gomory-Hu de G';
se *existe* $e \in E^T$ *tal que* $w_e^T < 2$ **então**
    para cada $e = \{s, t\} \in E^T$ *tal que* $w_e^T < 2$ **faça**
        seja S o conjunto das arestas de um $s - t$ corte mínimo em G';
        $\mathcal{D} = \mathcal{D} \cup \{\sum_{e \in \delta(S)} x_e \geq 2\}$;
    fim
senão
    se $x_e \in \{0, 1\}$ *para todo* $e \in E$ **então**
        //solução inteira e conexa é uma rota;
        seja $\mathcal{R}$ a rota formada pela sequência de clientes $(i_1, \ldots, i_k)$ imposta por x;
        se $\mathcal{A}((L_{i_1}, \ldots, L_{i_k}), B) = \emptyset$ **então**
            $\mathcal{D} = \mathcal{D} \cup \{\sum_{e \in \mathcal{R}} x_e \leq |V| - 1\}$;
        fim
    fim
fim
devolva $\mathcal{D}$.

Capítulo 6

Resultados Computacionais

O algoritmo proposto foi codificado em linguagem C, com a utilização do resolvedor de programação linear inteira IBM ILOG CPLEX 12.1. Foi utilizado o resolvedor de programação por restrições IBM ILOG CP Optimizer 2.3 para resolver a formulação em Programação por Restrições proposta na seção 4.3. As rotinas para encontrar cortes mínimos e cortes de Gomory Hu foram adaptadas a partir do código disponibilizado pelo ZIB [32]. Os algoritmos apresentados na seção 4.2 e 4.1 foram adaptados do código disponibilizado pelos autores. Os testes foram executados em um único core de 2.40 GHz de um Intel Core 2 Quad CPU Q6600 com 4GB de memória RAM e sistema Linux.

Geramos algumas instâncias aleatórias da seguintes forma. O contêiner tem dimensões sempre igual a $(233, 220, 587)$. A largura e a altura de cada item é algum valor inteiro aleatório entre 10% e 60% da largura e da altura do contêiner, respectivamente. A profundidade de cada item é um valor inteiro aleatório entre 5% e 40% da profundidade do contêiner. Cada item gerado é atribuído a um cliente de forma cíclica, tal que cada cliente tenha no máximo um item de diferença para qualquer outro. Foram geradas 2 classes de instâncias. Na classe "He" cada item é gerado independentemente dos outros, de forma que é esperado que não existam itens iguais. Na classe "Ho" os itens são gerados de forma que cada item possui em torno de 5 itens iguais a ele. Os clientes foram posicionados numa região de 100 x 100.

Na seção 6.1 apresentamos os resultados dos algoritmos de empacotamento, fornecendo uma rota qualquer pré-definida. Na seção 6.2 apresentamos os resultados do algoritmo *branch-and-cut* proposto no capítulo 5, utilizando diferentes estratégias de empacotamento.

Em todas as tabelas de resultados, a coluna $Classe$ representa a classe da instância. A coluna Vol representa a porcentagem do volume do Contêiner ocupada pela soma dos volumes das caixas. A coluna $Itens$ representa o número de caixas e a coluna Cli o número de clientes daquela instância.

6.1 Resultados para o Problema de Empacotamento

Na tabela 6.1 a coluna MPVBK$_{irr}$ apresenta o tempo necessário para o algoritmo MPVBK encontrar a solução ótima no caso irrestrito, quando não importa a ordem em que serão descarregados os itens. A coluna CP$_{irr}$ apresenta o tempo utilizado pelo algoritmo CP no caso irrestrito. Os tempos são reportados em segundos. Os campos indicados com "-" são aqueles onde os algoritmos não encontraram solução no tempo limite de 3600 segundos.

$Classe$	Vol	$Itens$	$MPVBK_{irr}$	CP_{irr}
He	36	17	0	0
He	50	21	0	0
He	64	25	1	67
He	65	26	1	36
He	67	27	2	28
He	69	28	701	173
He	76	29	3	-
Ho	46	17	0	0
Ho	51	25	6	3
Ho	52	21	0	1
Ho	61	29	271	46
Ho	68	31	737	1345
Ho	69	35	-	-
Ho	71	33	-	-
Ho	78	37	-	-

Tabela 6.1: Experimentos Computacionais: Caso irrestrito.

A tabela 6.2 apresenta os resultados obtidos pelos algoritmos no caso com ordem, onde uma rota pré-definida já é apresentada. As colunas CP, MPVBK e MPV apresentam o tempo em segundos decorrido para a resolução através dos respectivos algoritmos. A coluna CP_s apresenta o tempo utilizado pela abordagem CP sem a redução de domínio, ou seja, sem a utilização dos pontos de discretização. Colunas com "-" indicam que o algoritmo não encontrou solução em até 3600 segundos. Notamos que a abordagem CP obteve diversas soluções ótimas para instâncias nas quais o MPVBK e o MPV não foram bem sucedidos.

$Classe$	Vol	$Itens$	Cli	$MPVBK$	CP_s	CP	MPV
He	36	17	2	0	0	0	0
He	50	21	2	0	0	4	7
He	64	25	2	1	254	40	-
He	65	26	2	1	157	20	-
He	67	27	2	1	1647	65	-
He	69	28	2	84	815	133	-
He	76	29	2	-	-	-	-
He	36	17	4	0	0	0	0
He	50	21	5	0	23	1	9
He	64	25	6	0	70	25	-
He	65	26	6	1	168	12	-
He	67	27	6	-	456	41	-
He	69	28	7	-	-	108	-
He	76	29	7	-	-	323	-
He	36	17	8	0	1	0	0
He	50	21	10	0	33	1	2
He	64	25	12	0	2522	17	312
He	65	26	12	-	264	18	-
He	67	27	13	-	-	40	-
He	69	28	13	-	-	132	-
He	76	29	14	-	-	-	-
He	36	17	17	0	0	0	0
He	50	21	21	0	39	0	267
He	64	25	25	-	1567	16	-
He	65	26	26	-	-	14	-
He	67	27	27	-	-	190	-
He	69	28	28	-	-	56	-
He	76	29	29	-	-	-	-
Ho	46	17	2	0	1	0	0
Ho	51	25	2	5	52	3	0
Ho	52	21	2	0	73	0	0
Ho	61	29	2	231	377	9	-
Ho	68	31	2	309	-	904	-
Ho	69	35	2	-	-	-	-
Ho	71	33	2	-	-	-	-
Ho	78	37	2	-	-	-	-
Ho	36	17	4	0	0	0	0
Ho	51	25	6	2	32	1	0
Ho	52	21	5	0	18	1	0
Ho	61	29	7	68	-	17	-
Ho	68	31	8	-	-	79	-
Ho	69	35	10	-	-	706	-
Ho	71	33	9	-	-	-	-
Ho	78	37	11	-	-	-	-
Ho	46	17	8	0	11	0	0
Ho	51	25	12	2	230	0	0
Ho	52	21	10	0	30	0	0
Ho	61	29	14	4	-	4	-
Ho	68	31	15	-	-	37	-
Ho	69	35	17	-	-	756	-
Ho	71	33	16	-	-	1169	-
Ho	78	37	18	-	-	-	-
Ho	46	17	17	0	14	0	0
Ho	51	25	25	1	445	0	0
Ho	52	21	21	0	-	0	0
Ho	61	29	29	-	1991	10	-
Ho	68	31	31	-	-	53	-
Ho	69	35	35	-	-	580	-
Ho	71	33	33	-	-	1081	-
Ho	78	37	37	-	-	-	-

Tabela 6.2: Experimentos Computacionais: Comparação entre o CP, MPV e o MPVBK.

6.2 Resultados para o 3DTSPL

Na tabela 6.3 apresentamos os resultados obtidos para o 3DTSPLutilizando como sub-rotina de empacotamento somente a abordagem CP e também utilizando somente o algoritmo MPVBK. As colunas $T(s)$ indicam o tempo decorrido em segundos, a coluna $Cuts_{cplex}$ indica o número de cortes adicionados pelo CPLEX e a coluna $Cuts_{GH+pack}$ indica o número de cortes adicionados pela separação de conectividade e pela separação de empacotamento. Os tempos marcados com "-" indicam que o algoritmo ultrapassou o limite de 3600 segundos.

A tabela 6.4 apresenta os resultados obtidos para o 3DTSPL utilizando como sub-rotina de empacotamento MPV60, MPVBK60 e CP, ou seja, utilizando os algoritmos MPV e MPVBK como heurísticas antes da execução do algoritmo exato CP. Também apresenta o resultado utilizando o algoritmo MPV60, MPVBK60, e CP60, todos os tempos limitados e portanto o resultado não necessariamente será ótimo. As colunas $T(s)$ indicam o tempo decorrido em segundos, os tempos marcados com "*" indicam que o algoritmo terminou de executar e não conseguiu encontrar uma rota com um empacotamento. Os tempos marcados com "-" indicam que o algoritmo ultrapassou o limite de 3600 segundos. A coluna $Cuts_{cplex}$ indica o número de cortes adicionados pelo CPLEX e a coluna $Cuts_{GH+pack}$ indica o número de cortes adicionados pela separação de conectividade e pela separação de empacotamento. A coluna Gap indica a distância da solução heurística para a solução exata, em porcentagem.

Instância				*CP*			*MPVBK*		
Classe	*Vol*	*Itens*	*Cli*	$T(s)$	$Cuts_{cplex}$	$Cuts_{GH+pack}$	$T(s)$	$Cuts_{cplex}$	$Cuts_{GH+pack}$
He	36	17	2	0	0	0	0	0	0
He	50	21	2	0	0	0	0	0	0
He	64	25	2	23	0	0	1	0	0
He	65	26	2	34	0	0	1	0	0
He	67	27	2	96	0	0	2	0	0
He	69	28	2	309	0	0	-	0	1
He	76	29	2	-	0	1	-	0	1
He	36	17	4	0	0	0	0	0	0
He	50	21	5	0	0	0	0	0	0
He	64	25	6	38	0	2	-	0	3
He	65	26	6	69	0	1	1	0	1
He	67	27	6	39	0	1	-	0	2
He	69	28	7	65	0	0	-	0	1
He	76	29	7	-	0	4	-	0	4
He	36	17	8	0	0	0	0	0	0
He	50	21	10	0	0	0	0	0	0
He	64	25	12	117	0	1	-	1	3
He	65	26	12	27	0	1	2	0	1
He	67	27	13	195	0	1	-	0	2
He	69	28	13	537	0	2	-	0	3
He	76	29	14	-	0	4	-	0	4
He	36	17	17	0	0	0	0	0	0
He	50	21	21	0	0	2	0	0	2
He	64	25	25	41	2	10	2	2	10
He	65	26	26	172	1	10	-	1	11
He	67	27	27	33	4	10	-	5	11
He	69	28	28	1311	4	4	-	2	3
He	76	29	29	-	1	6	-	1	6
Ho	46	17	2	0	0	0	0	0	0
Ho	51	25	2	1	0	0	4	0	0
Ho	52	21	2	1	0	0	0	0	0
Ho	61	29	2	22	0	0	171	0	0
Ho	68	31	2	145	0	0	4	0	0
Ho	69	35	2	-	0	1	-	0	1
Ho	71	33	2	-	0	1	-	0	1
Ho	78	37	2	-	0	1	-	0	1
Ho	36	17	4	0	0	0	0	0	0
Ho	51	25	6	3	0	0	1	0	0
Ho	52	21	5	0	0	0	0	0	0
Ho	61	29	7	41	0	1	40	0	1
Ho	68	31	8	210	0	0	-	0	1
Ho	69	35	10	1532	0	2	-	0	3
Ho	71	33	9	-	0	1	-	0	1
Ho	78	37	11	-	0	1	-	0	1
Ho	46	17	8	0	0	1	0	0	1
Ho	51	25	12	4	0	2	1	0	2
Ho	52	21	10	0	0	3	0	0	3
Ho	61	29	14	45	1	2	17	1	2
Ho	68	31	15	88	1	2	-	1	3
Ho	69	35	17	1009	1	2	-	1	4
Ho	71	33	16	-	1	5	-	1	5
Ho	78	37	18	-	1	4	-	1	4
Ho	46	17	17	0	0	7	0	0	7
Ho	51	25	25	5	3	16	3	3	16
Ho	52	21	21	0	0	1	0	0	1
Ho	61	29	29	23	2	6	-	5	11
Ho	68	31	31	51	2	7	-	2	8
Ho	69	35	35	-	3	9	-	3	9
Ho	71	33	33	-	2	9	-	2	9
Ho	78	37	37	-	2	7	-	2	7

Tabela 6.3: Experimentos Computacionais: comparação entre CP e MPVBK para o 3DTSPL.

Instância				$MPV60$, $MPVBK60$ e CP			$MPV60$, $MPVBK60$ e $CP60$			
Classe	*Vol*	*Itens*	*Cli*	$T(s)$	$Cuts_{cplex}$	$Cuts_{GH+pack}$	$T(s)$	$Cuts_{cplex}$	$Cuts_{GH+pack}$	*Gap*
He	36	17	2	0	0	0	0	0	0	0
He	50	21	2	2	0	0	2	0	0	0
He	64	25	2	61	0	0	61	0	0	0
He	65	26	2	61	0	0	61	0	0	0
He	67	27	2	62	0	0	62	0	0	0
He	69	28	2	405	0	0	181*	0	1	-
He	76	29	2	-	0	1	181*	0	1	-
He	36	17	4	0	0	0	0	0	0	0
He	50	21	5	60	0	0	60	0	0	0
He	64	25	6	139	0	2	140	0	2	0
He	65	26	6	61	0	1	61	0	1	0
He	67	27	6	156	0	1	158	0	1	0
He	69	28	7	175	0	0	841	1	5	7,5
He	76	29	7	304	0	2	-	1	24	-
He	36	17	8	0	0	0	0	0	0	0
He	50	21	10	60	0	0	60	0	0	0
He	64	25	12	397	0	1	398	0	1	0
He	65	26	12	62	0	1	62	0	1	0
He	67	27	13	289	0	1	693	3	6	1,7
He	69	28	13	533	0	2	324	0	3	0,2
He	76	29	14	-	0	2	-	2	33	-
He	36	17	17	0	0	0	0	0	0	0
He	50	21	21	60	0	2	60	0	2	0
He	64	25	25	122	2	10	122	2	10	0
He	65	26	26	281	1	10	282	1	10	0
He	67	27	27	268	4	10	270	4	10	0
He	69	28	28	1614	4	4	2987	4	22	0,6
He	76	29	29	-	2	5	-	3	45	-
Ho	46	17	2	0	0	0	0	0	0	0
Ho	51	25	2	0	0	0	0	0	0	0
Ho	52	21	2	0	0	0	0	0	0	0
Ho	61	29	2	128	0	0	129	0	0	0
Ho	68	31	2	64	0	0	64	0	0	0
Ho	69	35	2	-	0	1	189*	0	1	-
Ho	71	33	2	-	0	1	185*	0	1	-
Ho	78	37	2	-	0	1	180*	0	1	-
Ho	36	17	4	0	0	0	0	0	0	0
Ho	51	25	6	61	0	0	61	0	0	0
Ho	52	21	5	0	0	0	0	0	0	0
Ho	61	29	7	102	0	1	102	0	1	0
Ho	68	31	8	305	0	0	342	0	1	0,8
Ho	69	35	10	1320	0	2	-	0	26	-
Ho	71	33	9	-	0	1	-	1	23	-
Ho	78	37	11	-	0	1	-	3	26	-
Ho	46	17	8	0	0	1	0	0	1	0
Ho	51	25	12	0	0	2	0	0	2	0
Ho	52	21	10	1	0	3	1	0	3	0
Ho	61	29	14	78	1	2	78	1	2	0
Ho	68	31	15	188	1	2	252	0	3	0
Ho	69	35	17	900	1	2	-	2	33	-
Ho	71	33	16	-	1	5	-	3	38	-
Ho	78	37	18	-	1	4	-	4	33	-
Ho	46	17	17	0	0	7	0	0	7	0
Ho	51	25	25	183	3	16	183	3	16	0
Ho	52	21	21	60	0	1	60	0	1	0
Ho	61	29	29	128	2	6	128	2	6	0
Ho	68	31	31	163	2	7	163	2	7	0
Ho	69	35	35	-	3	9	-	4	37	-
Ho	71	33	33	-	2	9	-	9	42	-
Ho	78	37	37	-	2	7	-	7	36	-

Tabela 6.4: Experimentos Computacionais: comparação do uso de heurísticas para o 3DTSPL.

Adaptamos algumas instâncias propostas em [11], e utilizadas também por [9] para o problema do Problema de Roteamento de Veículos com Restrições de Empacotamento Tridimensional. Nestas instâncias mantivemos a quantidade e posições dos clientes e do depósito, os itens de cada cliente, bem como a largura e a altura do contêiner. Como no Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensionalsomente um contêiner pode ser utilizado, flexibilizamos a profundidade do contêiner. Atribuimos duas profundidades tais que, o volume total dos itens ocupasse 60% e 70% do volume do contêiner. A largura do contêiner é de 25 e a altura é 30, e as dimensões dos itens variam entre 20% e 60% das dimensões do contêiner, a largura original é de 60.

A tabela 6.5 apresenta os resultados obtidos para o 3DTSPL utilizando como sub-rotina de empacotamento os algoritmos MPVBK e CP nestas instâncias adaptadas. A partir da instância 3l_cvrp18 com 94 itens e 44 clientes, nenhum dos métodos conseguiu resolver o problema no tempo máximo de 3600 segundos.

$Instância$			60%		70%	
$Nome$	$Itens$	Cli	$T(s)\ CP$	$T(s)\ MPVBK$	$T(s)\ CP$	$T(s)\ MPVBK$
MPVBK	32	15	10	326	28	-
3l_cvrp02	26	15	1	1	5	-
3l_cvrp03	37	20	6	32	85	-
3l_cvrp04	36	20	6	15	165	-
3l_cvrp05	45	21	136	-	939	-
3l_cvrp06	40	21	84	-	2538	-
3l_cvrp07	46	22	74	-	1272	-
3l_cvrp08	43	22	46	-	2197	-
3l_cvrp09	50	25	313	-	-	-
3l_cvrp10	62	29	294	-	-	-
3l_cvrp11	58	29	265	-	752	-
3l_cvrp12	63	30	450	-	-	-
3l_cvrp13	61	32	528	-	-	-
3l_cvrp14	72	32	1326	-	-	-
3l_cvrp15	68	32	1201	-	-	-
3l_cvrp16	63	35	174	-	3038	-
3l_cvrp17	79	40	1333	-	-	-
3l_cvrp18	94	44	-	-	-	-

Tabela 6.5: Experimentos Computacionais: instâncias da literatura adaptadas ao 3DTSPL.

Observamos que o método CP obteve um número maior de soluções ótimas, sendo que o método MPVBK ficou limitado as instâncias mais fáceis, com menos de 40 itens, e com ocupação de 60%. As instâncias 3l_cvrp18 até a 3l_cvrp27, que contêm entre 94 e 198 itens não foram resolvidas por nenhum dos métodos, e por isso foram omitidas da tabela. Notamos também que mesmo nas instâncias onde o método MPVBK obteve uma solução ótima, o método CP chegou a ser 96% mais rápido.

Denotamos as instâncias utilizadas da seguinte forma Class-Vol-Itens-Cli, por exemplo a instância He-76-29-7 corresponde a instância com 29 itens e 7 clientes, cuja soma dos volumes das caixas corresponde a 76% do volume do contêiner.

Escolhemos duas das instâncias de cada classe, a He-76-29-29 e a He-76-29-7 da classe He e as instâncias Ho-78-37-37 e Ho-78-37-11 da classe Ho. Essa escolha se deve ao fato de que essas instâncias apresentam uma grande quantidade de itens, e diversos clientes. Executamos sobre essas instâncias o algoritmo *branch-and-cut* utilizando as heurísticas 3HFF e GR como algoritmos de empacotamento, variando a profundidade do contêiner. Dessa forma podemos observar a diminuição do custo da rota conforme aumentamos o tamanho do contêiner.

As figuras 6.1, 6.2, 6.3 e 6.4 mostram o tamanho da rota quando variamos a profundidade do contêiner para as instâncias He-76-29-7, He-76-29-29, Ho-78-37-11 e Ho-78-37-37 respectivamente.

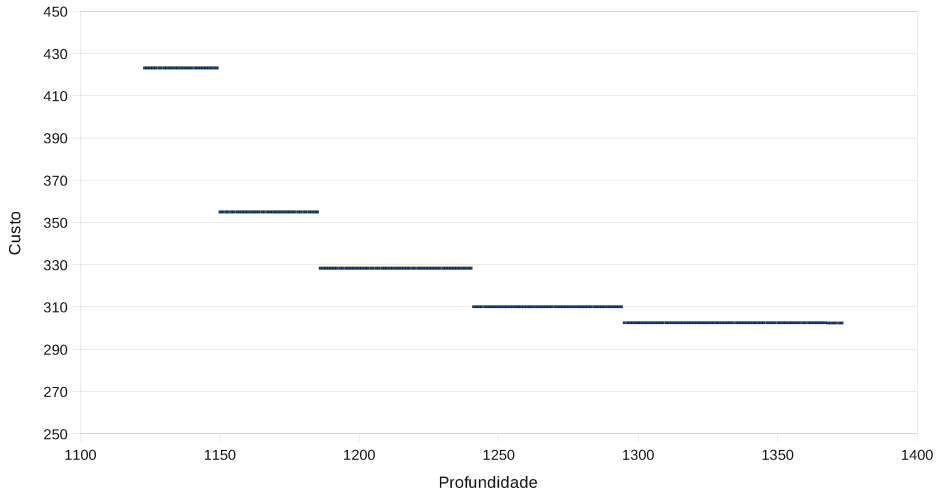


Figura 6.1: Experimentos Computacionais: Profundidade e Custo da Rota - He-76-29-7

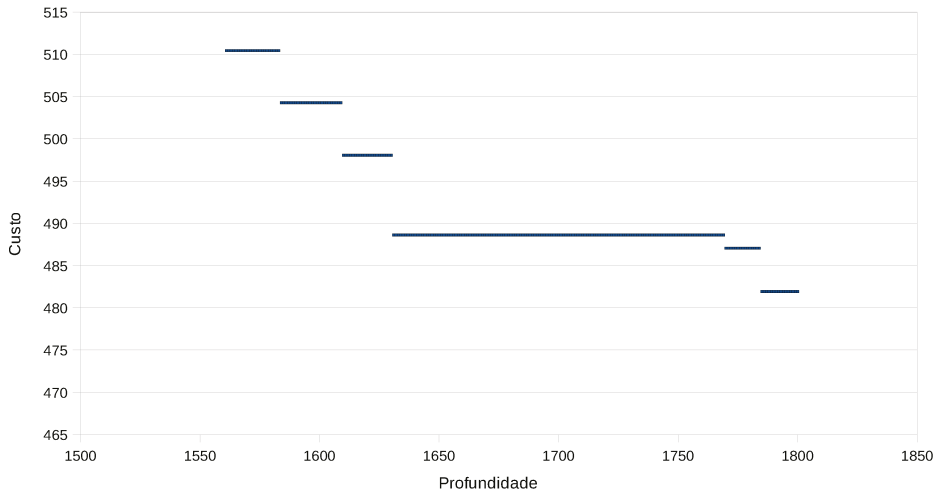


Figura 6.2: Experimentos Computacionais: Profundidade e Custo da Rota - He-76-29-29

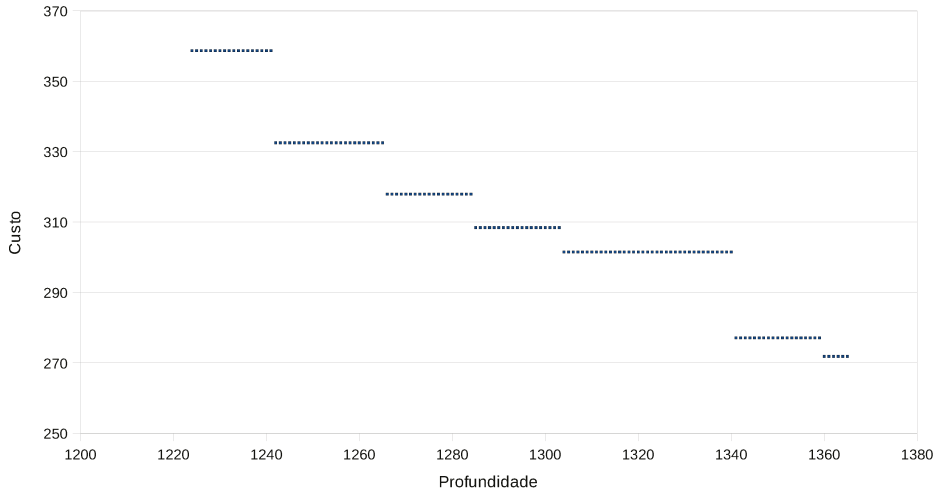


Figura 6.3: Experimentos Computacionais: Profundidade e Custo da Rota - Ho-78-37-11

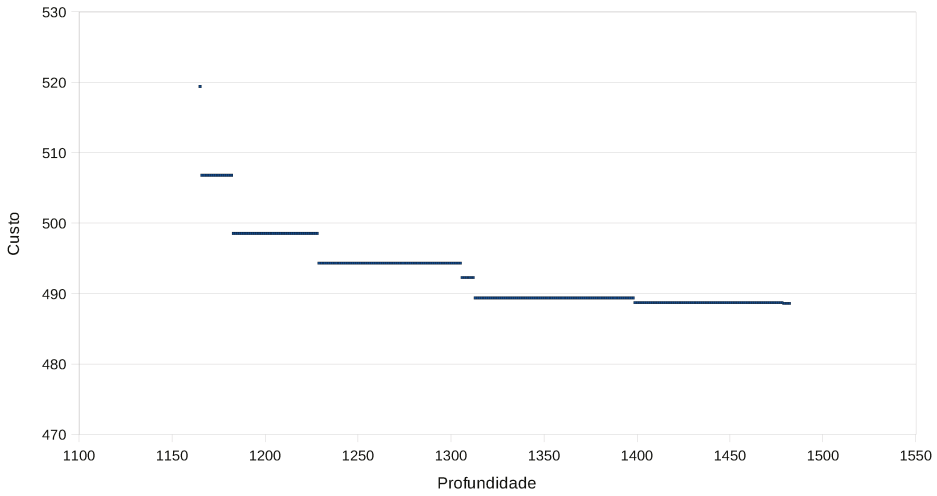


Figura 6.4: Experimentos Computacionais: Profundidade e Custo da Rota - Ho-78-37-37

Observamos que houve uma variação de quase 30% no custo da rota de He-76-29-7, enquanto o contêiner teve um aumento de aproximadamente 20%. Já na intância Ho-78-37-11 observamos uma melhora de aproximadamente 25% no custo da rota, com um aumento do contêiner de pouco mais de 10%. Esse tipo de análise pode ser muito importânte na tomada de decisões nos setores de logística.

Capítulo 7

Conclusões

Nesta dissertação apresentamos uma abordagem exata para o Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional. Utilizamos o método *branch-and-cut*, inserindo desigualdades válidas encontradas através de rotinas de separação. Duas classes de cortes são necessárias, os cortes de conectividade, que utilizam algoritmos de corte mínimo e método de Gomory-Hu, e os cortes de empacotamento, onde utilizamos algoritmos exatos, que se baseiam em *branch-and-bound* e Programação por Restrições.

Testamos o método em instâncias geradas aleatoriamente e obtivemos um bom desempenho para resolver instâncias de tamanho razoável. Em particular, verificamos que a abordagem CP foi superior no número de soluções ótimas obtidas.

Este resultado é bastante importante, pois verificou a viabilidade da abordagem CP mesmo com formulações simples, abrindo o horizonte para a adição de restrições redundantes que melhorem o desempenho do resolvedor. Também sugere que é possível a adição de novas restrições práticas, que tornem as soluções encontradas computacionalmente em soluções mais próximas as factíveis no mundo real.

Também estudamos o impacto da variação no tamanho do contêiner e o custo da rota, verificamos que uma profundidade de contêiner maior, pode levar a rotas de custo menor. Este resultado é interessante do ponto de vista logístico, pois muitas vezes a decisão de qual contêiner utilizar, tem algum impacto financeiro.

Além disso verificamos que a maior parte do tempo despendido na resolução de instâncias do 3DTSPL, é gasto para resolver o problema de empacotamento. Dessa forma é importante encontrar algoritmos eficientes para esse problema em particular.

As principais contribuições dessa dissertação foram algoritmos exatos para o Problema de Empacotamento Tridimensional com Restrição de Ordem. E uma abordagem exata para o Problema do Caixeiro Viajante com Restrições de Empacotamento Tridimensional.

Parte desse trabalho deu origem a um artigo [17] aceito na *XXXV Conferência Latino-Americana de Informática (XXXV CLEI)*. Nessa conferência apenas 29% dos mais de 460 arti-

gos foram aceitos.

Como trabalhos futuros, pretendemos investigar extensões naturais do problema, como o Problema de Roteamento de Veículos com Restrições de Empacotamento Tridimensional, e com outras restrições comumente consideradas na literatura.

Referências Bibliográficas

[1] V. A. Armentano and O. C. B. de Araújo. Carregamento de contêiner com restrições de múltiplos destinos. In *Anais do XLII Simpósio Brasileiro de Pesquisa Operacional*, page 10pgs, Bento Gonçalves, RS, 2010.

[2] D. S. Campos. *Integração dos problemas de carregamento e roteamento de veículos com janela de tempo e frota heterogênea*. PhD thesis, Universidade de São Paulo, Escola Politécnica-USP, São Paulo-SP, Brasil, março 2008.

[3] N. Christofides and C. Whitlock. An algorithm for two dimensional cutting problems. *Ops. Res.*, 25:30–44, 1977.

[4] J.-F. Cordeau, M. Iori, G. Laporte, and J. J. S. González. A branch-and-cut algorithm for the pickup and delivery traveling salesman problem with lifo loading. *Networks*, 55:46 – 59, 2010.

[5] O. C. B. de Araújo. *Problemas de corte e empacotamento tridimensional e integração com roteamento de veiculos*. PhD thesis, Universidade Estadual de Campinas - Faculdade de Engenharia Elétrica e de Computação, São Paulo-SP, Brasil, dezembro 2006.

[6] O. C. B. de Araújo and V. A. Armentano. A multi-start random constructive heuristic for the container loading problem. *Pesquisa Operacional*, 27:311 – 331, 08 2007.

[7] B. L. P. de Azevedo, P. Hokama, F. K. Miyazawa, and E. C. Xavier. A branch-and-cut approach for the vehicle routing problem with two-dimensional loading constraints. *XLI Simposio Brasileiro de Pesquisa Operacional*, 2009.

[8] E. den Boef, J. Korst, S. Martello, D. Pisinger, and D. Vigo. Erratum to "The Three-Dimensional Bin Packing Problem": Robot-Packable and Orthogonal Variants of Packing Problems. *Operations Research*, 53(4):735–736, July 2005.

[9] G. Fuellerer, K. F. Doerner, R. F. Hartl, and M. Iori. Metaheuristics for vehicle routing problems with three-dimensional loading constraints. *European Journal of Operational Research*, 201(3):751 – 759, 2010.

[10] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report, Carnegie-Mellon University, Pittsburgh, PA, 1979.

[11] M. Gendreau, M. Iori, G. Laporte, and S. Martello. A tabu search algorithm for a routing and container loading problem. *Transpor Sci*, 9(3):342–350, 2006.

[12] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[13] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *SIAM Journal on Computing*, 9(4):551–570, 1961.

[14] D. Gusfield. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing*, 19(1):143–155, 1990.

[15] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263 – 313, 1980.

[16] J. C. Herz. A recursive computational procedure for two-dimensional stock-cutting. *IBM J. Res. Dev.*, pages 462–469, 1972.

[17] P. Hokama, B. L. P. de Azevedo, and F. K. Miyazawa. Problema do caixeiro viajante tridimensional com restrição de ordem. *XXXV Conferencia Latinoamericana de Informática*, 2009.

[18] M. Iori, J. J. S. González, and D. Vigo. An exact approach for the vehicle routing problem with two dimensional loading constraints. *Transpor Sci*, 41(2):253–264, 2007.

[19] D. F. R. J. A. George. A heuristic for packing boxes into a container. *Computer and Operations Research 7*, 4:147–156, 1980.

[20] Z. Jin, K. Ohno, and J. Du. An efficient approach for the three-dimensional container packing problem with practical constraints. *Asia-Pacific Journal of Operational Research*, 21(3):279–295, 2004.

[21] L. Junqueira, R. Morabito, and D. S. Yamashita. Abordagens para problemas de carregamento de contêineres com considerações de múltiplos destinos. http://www.dep.ufscar.br/docentes/morabito/leonardo_gp09.pdf, 2010.

[22] K. K. Lai, J. Xue, and B. Xu. Container packing in a multi-customer delivering operation. *Computers & Industrial Engineering*, 35(1-2):323 – 326, 1998. Proceedings of the 23rd International Conference on Computers and Industrial Engineering.

[23] J.-L. Lin, C.-H. Chang, and J.-Y. Yang. A study of optimal system for multiple-constraint multiple-container packing problems. In M. Ali and R. Dapoigny, editors, *Advances in Applied Artificial Intelligence*, volume 4031 of *Lecture Notes in Computer Science*, pages 1200–1210. Springer Berlin / Heidelberg, 2006. 10.1007/11779568_127.

[24] S. Martello, D. Pisinger, and D. Vigo. The three-dimensional bin packing problem. *Operations Research*, 48(2):256–267, 2000.

[25] S. Martello, D. Pisinger, D. Vigo, E. D. Boef, and J. Korst. Algorithm 864: General and robot-packable variants of the three-dimensional bin packing problem. *ACM Trans. Math. Softw.*, 33, March 2007.

[26] A. Moura and J. Oliveira. An integrated approach to the vehicle routing and container loading problems. *OR Spectrum*, 31:775–800, 2009. 10.1007/s00291-008-0129-4.

[27] Q. Ruan, Q. Ruo, K. Woghiren, and L. Miao. A hybrid genetic algorithm for the vehicle routing problem with three-dimensional loading constraints. *Transportation Research Board*, 2011 - to appear.

[28] D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In A. Borning, editor, *Principles and Practice of Constraint Programming*, volume 874 of *Lecture Notes in Computer Science*, pages 10–20. Springer Berlin / Heidelberg, 1994. 10.1007/3-540-58601-6_86.

[29] L. Wolsey. *Integer Programming*. Wiley, 1998.

[30] L. A. Wolsey and G. L. Nemhauser. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1999.

[31] E. E. Zachariadis, C. D. Tarantilis, and C. T. Kiranoudis. A guided tabu search for the vehicle routing problem with two-dimensional loading constraints. *European Journal of Operational Research*, 195(3):729 – 743, 2009.

[32] Zib-Berlin. Mathprog: A collection of codes for solving various mathematical programming problems.
`http://elib.zib.de/pub/Packages/mathprog/index.html`.